REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
30-06-2006	Final	Dec 2001 - Mar 2006

4. TITLE AND SUBTITLE

Knowledge, Models, and Tools in Support of Advanced Distance Learning Final Report: The iRides Performance Simulation / Instruction Delivery and Authoring Systems

5a. CONTRACT NUMBER

5b. GRANT NUMBER

N00014-02-1-0179

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Allen Munro, Quentin A. Pizzini, Mark C. Johnson, Josh Walker, and David Surmon
University of Southern California Center for Cognitive Technology

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

UCLA CSE/CRESST
300 Charles E. Young Dr. North
300 GSE&IS/Mailbox 951522
Los Angeles, CA 90095-1522

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

ONR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report describes research conducted by the University of Southern California Behavioral Technology Laboratory in developing iRides, an advanced system for delivering authored interactive graphical simulations and instructional vignettes. The system provides the ability to deliver simulation-based instruction as Java applications, Java applets, and Java Web Start applications. The latter two options make it possible for these authored interactive graphical simulations and training to be delivered over the Web or any similar network to support advanced distributed learning. A new authoring tool, iRides Author, supports development of any simulations and instruction that iRides can deliver. The applet version of iRides can be delivered as a SCORM-compliant shareable content object. Another tool for authoring iRides simulations and training, Rivets, is a fast C++ program that has been compiled for three Unix-type operating systems: Linux, Silicon Graphics IRIX, and Mac OS 10.3 or later with X11. The flexible and open architecture of iRides makes it possible to employ this tool in collaboration with other advanced training system components, such as intelligent tutors.

15. SUBJECT TERMS

iRides, simulation authoring, advanced distributed learning

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(Include area code)*
Unclassified	Unclassified	Unclassified			

Knowledge, Models, and Tools in Support of Advanced Distance Learning

Final Report

UCLA Subaward No. 0070-G-CH640
Supported by Office of Naval Research Grant N00014-02-1-0179

Allen Munro
Quentin A. Pizzini
Mark C. Johnson
Josh Walker
David Surmon

Behavioral Technology Laboratories
University of Southern California
250 No. Harbor Drive, Suite 309
Redondo Beach, CA 90277

(310) 379-0844
munro@usc.edu
http://btl.usc.edu/

Final Report

Knowledge, Models, and Tools in Support of Advanced Distance Learning

UCLA Award 0070-G-CH640
Supported by Office of Naval Research Grant N00014-02-1-0179

1.0 A technology for delivering simulation-based distance learning

1.1 Instruction in the Context of Simulations

Conventional distance learning is typically based on a *page* model of instruction. A student reads material on a page and/or views some graphics or listens to audio associated with that page, and then takes some action that leads to the presentation of another page. In some cases, a book-like model is used; students 'turn the pages' by pressing Next and Back buttons. In other learning modules, students may answer questions about the content that has just been presented to them. Depending on their responses, they may progress to different pages: one if they answer correctly, different ones that provide remedial instruction if they answer incorrectly.

This conventional approach may be adequate for many topics. This is unsurprising, since people have learned effectively from page-based books and other printed materials for hundreds of years. The gate-keeping function of the mini-assessments that permit the students to move through the learning materials make it possible for some students to move more quickly through such a computer-based set of pages than they might through a textbook. Certain other students, who might not learn effectively from a textbook with too condensed a presentation approach, benefit from the remedial materials that are built into certain well-designed page-based training systems.

Many subject matters can be effectively taught with well-designed and well-executed page-based training systems. These include subjects like history, literary theory, and perhaps the structure of organizations, devices, and other systems, Certain other subject matters have also often been taught using page-based approaches, but are, in fact, less well-suited to those methods. These include topics such as complex procedures, problem-solving strategies in specific contexts, and the behavior of organizations, devices, and other systems. For many of these types of topics, an action-centered learning context is much more effective than a page-based approach. For distance learning, the only practical method of delivering action-centered learning contexts is to use interactive graphical simulations.

This is not to say that delivering a topic-relevant interactive graphical simulation is a sufficient means of conveying knowledge about the topic. Interacting with a simulation is not enough to ensure that learning has taken place. Students typically require instructional control, guidance, and assessment. Without such interventions, only the

best-prepared and most highly motivated students are likely to learn much simply by interacting with graphical simulations.

1.2. Learning complex job skills

Adult learning is often motivated by the need to learn how to perform complex tasks that are important to the performance of one's job. In some cases, the procedures that must be learned to perform these tasks can be defined as a sequence of well-specified actions. In other cases, the procedure can be thought of instead as a network of conditional branchings that associate collections of actions that could be performed to accomplish the task goal in a variety of contexts. And in some cases, tasks are complex enough that they cannot be taught in terms of predefined observations and actions, but must instead be represented as a set of strategies, tactics, and heuristics that are sufficiently abstract that they can be applied to new circumstances that cannot be anticipated by the developers of the distance learning materials.

Whatever the complexity of the procedural or applied problem solving subject matter, practice at the task under a variety of conditions will improve the chances that the learner will apply what has been learned to actual tasks.[1] Most procedures require learning about sequences of actions, learning about the conditions under which actions could be taken, or learning how to address complex problems in the context of a task environment. These cognitive components of procedural learning can be taught in a practice environment that provides an opportunity for interactive practice in the context of a functionally realistic task environment, such as a graphical simulation. When a computer delivers the training, it is possible for it to tirelessly observe and offer pedagogically relevant responses to the actions that students take in that practice environment. Task practice can be interactively assessed in real time, rather than being postponed for 'after-action reviews', as is common in simulation-based training in the U.S. military forces.

A body of work on the topic of machine-based teaching and tutoring in the context of interactive simulations has produced two sets of specifications that may have wide application to the field of teaching cognitively loaded tasks using computers. One of these is a set of techniques that have been found effective for organizing and delivering automated instruction about procedures and complex tasks. The other is a set of specifications that action-centered environments, whether they be simulations or embedded training systems, can adhere to in order to support machine observation and remediation of learner actions during task practice. These results were applied to distance learning in the course of this project.

Procedural instruction has much in common with other types of instruction, such as the need for frequent assessment and for customized pedagogy. In addition, however,

---

[1] Certain types of procedural skills cannot be taught economically using distance learning technology at this time. In particular, where specialized motor skills must be learned (applying a precise twisting force during an assembly process, for example), distance learning cannot address the entire learning requirement. Cognitive components of procedural knowledge *can* be taught with distance learning, however.

procedural instruction has a number of characteristics that are not found in other types of instruction. Procedural instruction can include specialized *modes of instruction* and specialized *types of assessment.*

Procedural instruction is often carried out in a master-apprentice context. A tutor, who knows very well how the procedure should be performed and (ordinarily) why each action should be taken, works to convey that knowledge to a student or trainee. In many cases, procedural training is carried out in a work environment, which makes it possible to demonstrate actions and procedures, to point out relevant observations that should be made, and to assess student knowledge by observing the student carry out sequences of actions. As is described below, it is also possible to develop and deliver distance-learning procedural instruction that performs the same kinds of functions.

1.2.1. Special Instruction Modes. Consider the following possible major modes of instruction in procedural training:

- Demonstration
- Monitored practice
- Unmonitored practice

These modes are in addition to conventional instructional modes such as presenting explanations (in text, graphics, or animations) or assessing students' answers to questions.

1.2.2. Special Types of Assessment. A type of assessment that is found in procedure training is a *procedural performance assessment.* In a procedural performance assessment, a student is directed to carry out a procedure in one or in a variety of contexts. The student's approach to the procedure is observed and assessed. Detailed assessments can be used to guide instruction. Summative assessments can be used to determine whether a student can be credentialed as a competent performer of the targeted procedure.

A procedural tutor can carry out a set of *micro-assessments* during training. Micro-assessments are simple interaction events that can be assessed to provide estimates of knowledge about particular facts and skills that are required for competence in the procedure as a whole. Examples of micro-assessments can include

- Observing actions (in a procedural environment; here we are discussing actions that may be part of the procedure to be learned, not actions such as answering a multiple choice question)
- Observing effects (in the procedural environment)
- Observing the attainment of states of interest in the procedural environment

Just as a human tutor must observe actions, pose questions, explain relationships, carry out actions and sequences of actions, set up practice contexts, and pose challenges to be met by a learner, so, too must a web-delivered training system.

1.3. Teaching procedural knowledge in the context of a simulation

During the 1980s there was an enthusiasm for so-called *microworlds* as learning environments (Papert, 1980; Forbus, 1984; Hollan, Hutchins, and Weitzman, 1984). It was widely expected that providing students rule-governed interactive graphical environments would afford the students opportunities to acquire deep understandings of these rule-based systems, which they could interact with, conducting 'experiments' and observing the outcomes. Microworlds in topics such as microeconomics (Shute and Glazer, 1990), gravity and Newtonian motion, and electronic circuits were developed. In the long run, the microworld approach was not much applied to real world procedural training. There appear to be two primary reasons for this. The first is that developing high-quality microworlds for learning proved to be a rather expensive programming problem. Each microworld was typically created from scratch in a fairly low-level development environment (typically, using a programming language). There were substantial expenses re-implementing core simulation features and student interaction interfaces anew for each such project.

The second reason for the lack of ubiquitous success with microworlds is that it turns out that not every student has the intellectual and emotional makeup that drives them to experiment boldly, relentlessly, and imaginatively in the microworld. It is probably not a coincidence that one of the few surviving centers of educational microworld research is at MIT, where there is no shortage of students who exhibit such characteristics. To expect every electronics technician to derive or intuit important electronic principles as a result of experimenting with simulated electronic environments seems naïve, at best. In fact, subsequent work has found it necessary to supply pedagogical interactions with microworld simulations (Rieber, 1992; Self, 1995).

Similarly, to expect every schoolchild to derive the basic principles of velocity and acceleration as a result of using (real or simulated) ramps and rolling balls of different masses but equal rolling resistance is simply unrealistic. After all, people had access to ramps and balls to experiment with for thousands of years before Galileo worked out the basic principles of acceleration using such an apparatus. And he did not do so in the course of a single two-hour 'discovery learning' session. Not every student is a Galileo. Most people need guidance if they are to learn about the behavior of complex systems and about how to carry out procedures in those contexts. For most students, exposure to a microworld by itself will never bring about such learning. Students need to receive instruction in the procedural context.

What does instruction typically consist of in an interactive procedural context? There are many possible sequences of instruction, and many types of 'instructional primitives' that can be used in such sequences. Some of these instructional primitives will be of types that are commonly used in non-procedural instruction. For example, a teacher (or an artificial tutor) could begin by posing a simple yes-or-no question, such as "Have you ever seen an electronic heart defibrillator, such as this one?" In this paper, the topic is the types of instructional interactions that take place in procedural learning contexts, but that are not used in many other contexts, such as conventional page-based CBT/CBI/CBL.

1.4. The architecture of complex procedural learning environments

Whether instruction about procedures involves a human tutor and real world systems and devices, or an artificial tutorial component and a computer simulation, essentially the same systems architecture can be described. See Figure 1.

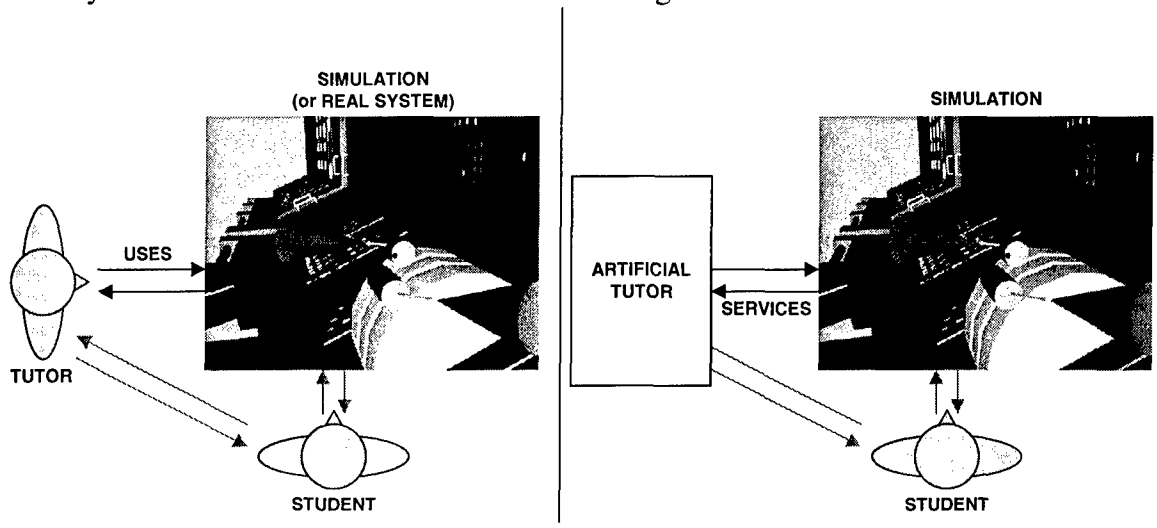


Figure 1. Pedagogical Services of Simulations for Tutors

We can describe the uses that a human tutor makes of a simulation or a real system during procedural training as requiring certain *services* of the simulation. Simulations must provide a similar set of services to an artificial tutor, if that tutorial component is to make effective use of the simulator for training. While it may seem odd to speak of a real system providing the service of permitting that the tutor manipulate one of its controls, a systematic approach to describing the interactions between tutors and simulations (or real systems) makes it possible to identify universals of procedural instruction.

During procedural training, many types of instructional interactions take place. Some of these are not unique to procedural instruction, but can be found in many types of education and training that do not involve interactions with behaving systems (simulations or real systems). Examples of such instructional interactions include verbal explanations, questions to the student and the corresponding answers, and so on. In Figure 1, these types of interactions are represented by the light gray arrows between the tutors and the students. The focus of this discussion is the types of instructional interactions that make direct use of features of the behaving environment. In particular, we are interested in identifying the simplest types of instructional interactions, which we call *instructional primitives*. More elaborate types of frequently observed instructional interactions are composed of sequences of such simpler types. The details of these instructional primitives for complex procedural learning are presented in Munro, Surmon, and Pizzini (in preparation), which also describes the underlying services that a simulator must deliver in order to support these types of instructional interactions. Here is a simple summary list of the major instructional primitives:

- Require Indication—The student is required to point to a simulated object, and receives progressively more explicit prompts to enable him or her to do so.

- Draw Attention to—An object is made visually prominent (perhaps by flashing a bounding box around it).
- Set Value(s)—The instruction changes some aspect of the interactive simulation by setting one or more values of attributes of the simulation.
- Perform Manipulation—The instruction changes the simulation in the same way that the user could, by carrying out a simulated action.
- Pause (Resume) Simulation—Freeze the simulation. And, when desired, start it up again as though no time had passed.
- Require Manipulation—Require that the student user perform a specific action. Permit no other actions, and provide progressively more explicit remediation to guide a correct response.
- Require State—Require that the student achieve a specific state of the simulation. Guidance is less controlling, more patient.
- DoMagic—Invoke some special capability of the simulator. This is a powerful feature that often proves useful, but it is not used for describing ordinary instructional uses of the simulation.

1.5. iRides

iRides is general-purpose environment for teaching in the context of simulations using the instructional primitives described above. *iRides* is a java program (which can be delivered as an applet, as a java application, or as a WebStart application) for delivering interactive graphical simulations and instruction that works in the context of such simulations. In its applet and WebStart forms, it can be used to deliver simulation-centered instruction over the Internet or local networks. It has two major components, a simulation engine that interprets simulation specifications in order to provide the appearance and the behavior of a simulated environment, and an instructional engine that interprets training specifications to deliver instructional interactions in simulation contexts. The first iRides system was developed under Office of Naval Research funding, which supported the development of the iRides simulation engine, and Air Force Research Laboratory funding, which sponsored the development of the original iRides instructional engine. During the course of the present sponsored research project, iRides was significantly revised, modified, and extended. The most important of these enhancements are described in Section 2, below.

In addition, during the course of this project we developed a Java-based authoring tool, *iRides Author*, that can be used to create simulations and instruction that iRides can deliver. IRides delivers interactive graphical simulations that are specified by text files with a ".jr" file format. IRides instruction is specified by XML text files called ".lml" files, which make use of the data type definitions in LessonML.DTD files. Before the advent of iRides Author, the ".jr" simulation specifications and ".lml" lesson specifications could be authored either by typing them in a text editor or an XML editor, or they could be generated by recent versions of the VIVIDS/RIDES program.[2] In other

[2] RIDES and its successor VIVIDS are C++ programs that were developed for Unix platforms, including Sun OS, The Silicon Graphics Unix implementation called IRIX, and early Santa Cruz

words, before iRides Author, one could port a Classic VIVIDS product for delivery by iRides, or one could use a text editor.

IRides Author is also designed to support the delivery of instruction under the control of authorable high level instructional tactics and strategies, as described below.

2.0 Authoring simulation-based distance learning with *iRides Author*

iRides Author supports both simulation authoring and instruction authoring. Simulation authoring has two major components: creating graphical objects, and specifying how those objects should interact with each other and with users.

2.1. *Creating simulation graphics*

Simulation graphics for a particular simulation can be created in a variety of ways: they can be imported from libraries of simulation objects, they can be drawn, or they can be imported as images obtained from scans, photos, or other sources. No matter how the graphics are added to a simulation, they can be assigned a variety of interactive behaviors using the iRides simulation language, as will be described below.

2.1.1 *Importing simulation objects.* Previously authored simulation objects can be imported in iRides Author, using the Insert command from the Edit menu. Choosing this command opens a file-picking interface that can be used to select an iRides object file for insertion into the present simulation. The selected object is placed in the topmost simulation window. If an object has defined (previously authored) behaviors, those behaviors will be imported along with the object's graphics.

2.1.2 *Drawing objects.* An author can draw an object by selecting one of four drawing tools. In Figure 2, the leftmost option is the selection/simulation tool. This tool is used to select graphic objects and to interact with them as a student/user would. The four tools to the right are the rectangle tool, the ellipse tool, the line tool, and the multi-line tool.

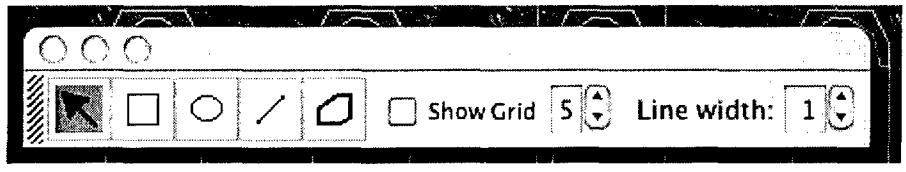


Figure 2. Drawing Tools in iRides Author

The first three of these tools are used by dragging the mouse from one point to another. The multiline tool is used by carrying out a series of clicks to set vertices, double-clicking to end the process of drawing an object. Authors can activate a grid (see Figure 3) that will constrain drawing actions to the intersections of the grid.

Operations Unix. VIVIDS/RIDES could be used to develop interactive graphical simulations and training scripts that worked in collaboration with these simulations.

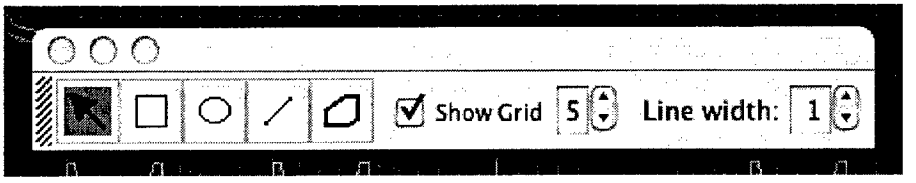


Figure 3. Drawing Grid

The author can set grid spacing. The *Line width* control determines how thick the lines are in a drawn graphic object.

2.1.3 *Importing images.* Another type of simulation graphic is an image. IRides can display images that are imported from jpg or gif files. Authors can create image files using digital cameras, scanners, and graphic editing or authoring tools, and then use these images as simulation objects. Each such image is a single, unitary graphic component of a simulation; it can be assigned certain interactive behaviors, but the simulation cannot make different parts of the image behave differently. The image objects as wholes can be made to move, rotate, scale themselves, or disappear and reappear.

2.1.4. *Grouping graphical objects.* In a sense, there is a fourth type of graphic object in iRides, a group whose members can include any mix of any of the types of graphics (including, of course, other graphical groups).

2.2. *Graphical attributes and graphical behaviors*

2.2.1. *Four universal graphical attributes.* One aspect of interactive behavior in an iRides simulation is appearance changes. The way that a given object looks at any point during a simulation is largely under the control of its *graphical attributes*. Every graphical object, no matter what type, has four attributes: Location, Rotation, Scale, and Visibility. Motion is simulated by changing the value of the Location attribute. The orientation of an object is controlled by the value of its Rotation attribute. The Scale attribute is used to determine how stretched or compressed the graphic is. The value of the Visibility attribute controls whether an object can be seen at a given time or not.

2.2.2. *Controlling graphical behavior.* Attributes can be given rules that determine the value of an attribute. For example, if this rule

```
if MouseDownIn(self)
   .sys.MousePosition
```

is assigned to a Location attribute, it means that when the left mouse button is held down while pointing to the object, the object's Location attribute will be set to the position of the mouse. (`.sys.MousePosition` is a special attribute that always holds the current location of the mouse pointer whenever the mouse button is down.) As a consequence of this rule, as a user drags the mouse with the button down, the object will move along with the mouse pointer. This simple rule can make any graphical object a user-moveable object.

2.2.3. *Simple graphical animation.* Similarly, if an author has drawn a line that is to be the second hand of a simulated wall clock, a simple rule can be used to ensure that the hand will turn at the rate of one cycle per minute. There are 60 seconds in one minute of

time, and there are 360 degrees in one complete rotation. Therefore, the second hand must rotate 6 degrees every second. This can be controlled by a simple rule for the Rotation attribute of the second hand that refers to the special attribute **.sys.Clock**, which always holds a time value, expressed as a number of seconds.

```
.sys.Clock * -6
```

This rule will update the rotation continuously as the clock value changes. The value of **.sys.Clock** changes many times per second, so the second hand will rotate continuously. (The reason that the "-6" factor is negative is to cause rotation to the right, at a rate of 6 degrees per second.) If the author wants to simulate the type of clock second hand that moves only once per second, the rule can use the **trunc()** function, which returns only the whole number part of a real number.

```
trunc(.sys.Clock) * -6
```

Note that both the earlier example rules for controlling a Location value and the rules for controlling the Rotation of a clock's second hand exhibit animation, but that the author does not have to be concerned with managing a repeating loop that controls when these rules will be applied. Each is automatically applied whenever it can be. For example, a rule that refers to the value of .sys.clock will be applied whenever the clock value changes. This automatic updating is one of the advantages of using iRides for simulation delivery, rather than using a programming language that requires creating all the simulation control independently in each simulation.

2.2.4. *Graphical attributes for objects with lines.* The four basic drawing tools (rectangle, ellipse, line, and multi-line) create graphical objects that all have intrinsic attributes that control the color, style (dashed, etc.), and width of the lines. The names of these attributes are PenColor, PenStyle, and PenWidth. Three of the four basic drawing tools (rectangle, ellipse, and multi-line) also have interior areas that can have color or patterns. The attributes that control these graphical characteristics are FillColor and FillPattern. One of the options not available in VIVIDS is to use graphics files as sources for textures. The author can specify in FillPattern what image file should serve as a source for the texture of a graphic object. It is possible to make the colors and other graphical characteristics of any of the basic drawing tool objects change in response to user events or in response to changes in simulation attribute values, simply by writing rules that determine the values of their intrinsic graphical attributes.

2.2.5. *Tools for color selection.* Authors can use color selection tools to determine the colors of lines and fill areas. See Figure 4, below

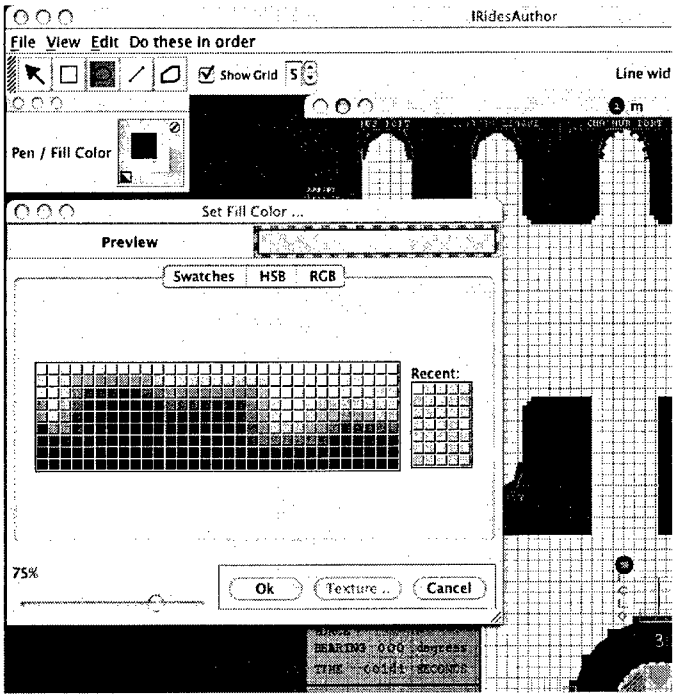


Figure 4. Drawing Tools in iRides Author

In addition to selecting pen color and fill color with this interface, authors can choose the transparency of the pen and fill areas for graphics that have the PenColor and FillColor attributes. Transparency can also be controlled by authored simulation behavior rules.

2.3 *Tools for Authoring Behavior*

As was mentioned above, the behavior of attribute values can be specified with relational rules, a type of one-way constraint programming (Munro, 2003). In the course of this research program, a set of behavior-authoring interfaces that had been partially developed previously was completed, debugged, and tested.

2.3.1. *Extended simulation language.* The iRides simulation language was extended significantly in the course of this project. The full language as of May 2004 is documented in *The iRides Simulation Language* (Munro, Pizzini, and Johnson, 2004). That document is included as Appendix A to this report.

2.3.2. *Integration with CRESST data services.* The new simulation function *putURL* supports sending data to the CRESST database. Thanks go to Matt Zhang at CRESST for support on this effort.

The CRESST Concept Mapper was also integrated with iRides. It is treated as a special type of Query. A student's concept map is submitted over a network connection to an Oracle-based CRESST assessment system that returns a number that represents the 'expertness' of the submitted map. This value is the Query result, and an LML lesson can refer to it to determine a subsequent course of instruction.

2.3.3. *Simulations with COTS graphics.* A special version of iRides was created that uses an SVG (Scalable Vector Graphics) renderer. With this version of iRides, one can use Adobe Illustrator or CorelDraw to create a simulation scene. By following appropriate naming conventions for the SVG objects, a simulation author can write iRides rules that control aspects of the appearance of the SVG objects. At present, this version of iRides, called *siRides*, has certain limitations, such as unexpected rotation behavior. (All graphics have their origin in the lower left corner of the 'page' or scene.) Additional work would be required to make siRides a competitive alternative to standard iRides.

2.3.4. *Behavior authoring aids.* In early releases of iRides, copying and pasting was only possible for text. It can now be performed on attributes and events. The Cut, Copy, Paste and Delete operations can also be undone and redone.

An aid to tracing the rules that refer to an attribute has been developed.

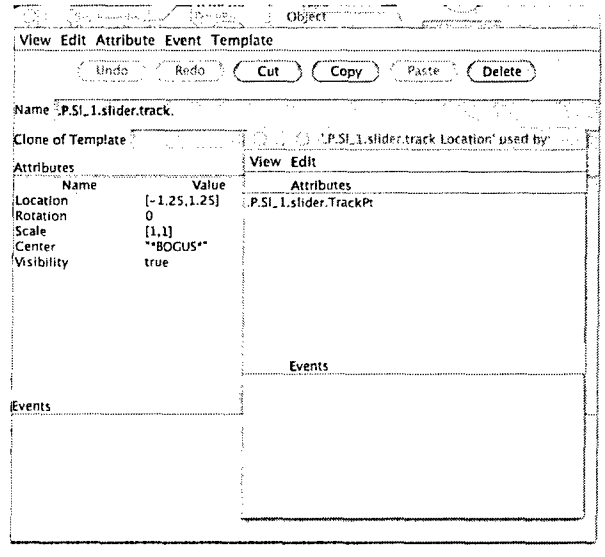


Figure 5. The 'Used By' Command Opens a List of Attributes that Use This Attribute

The 'Attribute' menu of the object data view has a command, 'Used By'. When this command is issued while an attribute in the object data view is selected, an interface opens that lists all of the Attributes and Events, if any, that refer to the selected attribute. Data views of those attributes and events can then be opened from that 'Used By' interface.

A toolbar with icons can be used to carry out frequently used operations such as Undo, Redo, Cut, Copy, Paste, and Delete.

A right-button menu was added to iRides Author to support opening object data views and carrying out other common operations. If the properties file for iRides (btl.prp) includes

graphics.authoring=true

then a double-click in a simulation object brings up the editable data view of that object. There is no need to go through the two-step process of right clicking on the object to

bring up a menu and then selecting a command from the menu to open the object's data view. However, if the object is part of a hierarchy of objects, the view that is opened may not be the one desired. For grouped objects, using the right-button menu makes more sense, because it offers access to all the objects in the hierarchy.

2.3.5. *Debugger.* An iRides Author debugging tool was added to the authoring application. It allows authors to trace executions by stepping through the executions of events and of relational rules. The debugger can show the text of events, including the comments, and can accurately point to each statement as it is executed in turn by Stepping. In Figure 6, the pointing hand icon at the left indicates which line of the iRides Event will be carried out next, when the Step button is pressed. (Clicking on Step Out would complete the entire event without stepping through the remaining statements individually.) Authors can set break points on relational rules and on events.

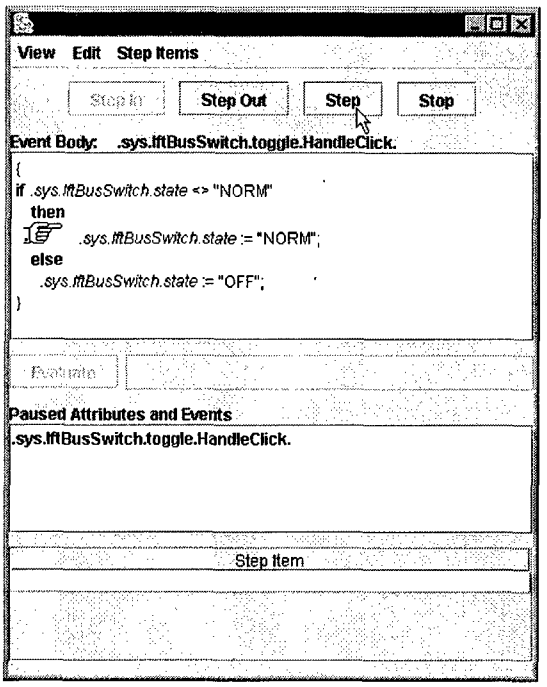


Figure 6. Debugging Window with Formatted Simulation Rules

The Step In and Step Out features were not present in earlier BTL debugging tools. When an author is stepping through the execution of an event and a call to another event is encountered, the Step button will cause the statement with that call to be evaluated in its entirety, and the debugger will advance to the next statement. If the author instead chooses the Step In button, the debugger will display the first line of the called event/function. Authors can then step through the statements of that function. When execution is stepping within an event, an author can choose Step Out, which completes the execution of the current event and then pauses again at the next executable statement or relation.

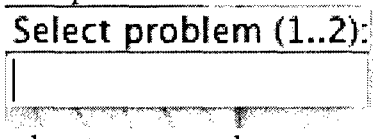
2.3.6. *Cloning objects.* iRides objects can be cloned under rule control. Each clone of an object has the same values and rules as the template object that it was created from. The values can be altered under the effects of the interactive simulation. Cloned objects can

also be deleted. The addition of cloning significantly extended iRides capabilities beyond those of RIDES, because simulations with indefinite numbers of transitory objects are now possible. This feature was used effectively in the Engineering Duty Officer's Data Analysis Tool simulation, described in Section 3.2 of this report.

2.3.7. *Authoring tools in support of cloning.* A user of iRides Author can create a new clone template by selecting an ordinary object and directing that a template be made from it. When this happens, the template is created, and the object is automatically designated as the first clone of that template. Users have several ways of selecting a template for editing, as well.

Authors can create new clones interactively in iRides Author, using a menu command in the template data view. Clones can also be created under rule control.

2.3.8. *'Native' user interface elements in iRides simulations.* New iRides simulation language functions will allow an author to present standard user interface components, rather than requiring that such components be reinvented in the iRides simulation language. For example, the *Text Entry Dialog* function makes it possible to use a 'native' text entry object, such as the one depicted here:



This user interface has several advantages over the one created using the primitives of the iRides simulation language.

- A blinking insertion point
- Selectable text
- Deletion of selected text
- Typeover of selected text
- A 'native' look and feel

Similarly, popup menus, which can have submenus, can be added to a simulation, using an iRides simulation call that invokes the native Java Swing menu interface object.

2.4 *Tools for Authoring Instruction*

2.4.1. *The LML language and its interpreter.* IRides lessons are specified in XML files that include reference to the Lesson Meta Language data type definition. An iRides instruction interpreter reads these instructional specifications and interactively delivers them to students. In the course of this project, two phases of development of the LML interpreter have occurred. In the first phase, the original iRides interpreter (developed with the support of the Air Force) was debugged and utilized. In the second phase, this language was revised and simplified, and the LML interpreter was revised to accommodate these changes in the instructional specification language. Thread locking problems were thereby eliminated, and the lesson delivery system is now reliable, as well as being flexible and powerful.

2.4.2. *Customizable user interface for instruction.* IRides has been given a number of customizable features for student interfaces, in addition to the complete customizability of the simulation. One of these is that the command buttons ('Continue', 'Help', etc.) can appear in a variety of locations and can have text labels or icons. The choice of members of the set of these buttons is completely up to the author. Another area of customizability is the interface for presenting textual instruction. This interface can be within the simulation window, or above, below, or to either side of it. It can present either simple text or HTML.

2.4.3. *Micro-assessment interfaces.* During training, it is frequently necessary to check that a student has understood a presented concept or fact, before proceeding to new material or before providing practice in applying the concept or fact. The iRides Query system was extended in a number of respects. Query windows have been made floating windows, so that they cannot be accidentally buried under simulation windows or presentations. Alternatively, authors have the option of specifying that queries be embedded in the window that presents instructional text. Each of these options is useful in certain training circumstances.

Each type of Query can have an associated authoring template that specifies the interactive structure of that type of student-tutor interaction. Each such template can have an associated graphical user interface for specifying the content of a particular Query. For example, by using a 'read indicator' instructional template, an author can specify a few property values that determine the content of a particular set of 'read indicator' query interactions. The lesson will ask the student to find a particular object and identify the value of one of its attributes. This interaction will be placed within a loop so that the student can have more than one chance to produce the correct response. A lesson authoring dialog in iRides author supports the specification of this type of instructional interaction without requiring that the author be able to read and edit XML.

A *table* Query interface and a page entry interface for Queries were implemented. These allow the presentation of a set of questions from a single GUI. In the table interface a table is shown with any number of cells. The cells may display an initial value. When the user clicks on one of the cells, a simple Query GUI appears, such as a text entry or a radio button entry. When the user responds to that Query, the response is recorded in the cell. An OK button is present in the table interface; it is clicked on when the user is satisfied that all the cell entries have been satisfactorily responded to.

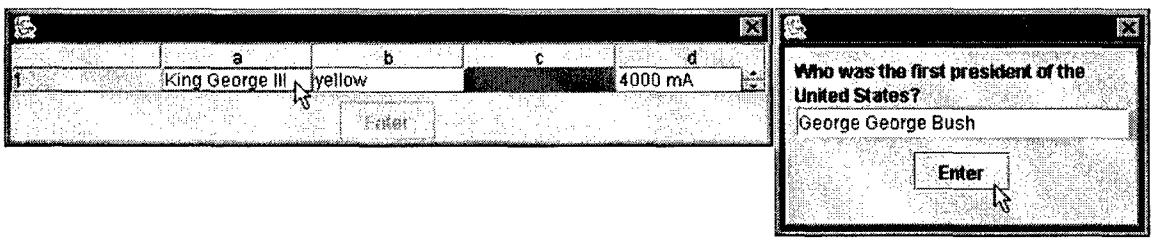


Figure 7. Answering a Text Question in a Table

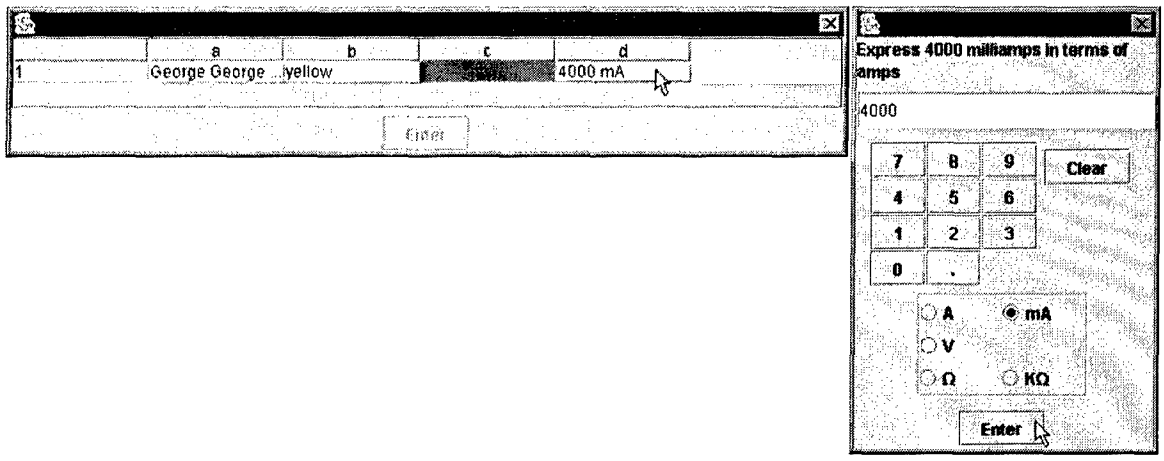


Figure 8. Answering a Numeric Question in a Table

As with simpler questions, an author can control many aspects of the layout and structure of questions using the LML specification format. A dialog interface in iRides Author simplifies the production of these specifications without manually writing the necessary LML text.

The *page* Query allows a more direct representation of the set of queries. It might display one or more text Query interfaces, or radio buttons, or check boxes. For numeric Queries, the user clicks on what looks like a text Query interface to bring up a keypad interface. An OK button is present in the page interface; it is clicked when the user is satisfied that all the entries have been satisfactorily responded to. Additional features were added to iRides to support the recording of student performance data in micro-assessments.

2.4.4. *GUIs for authoring instruction.* Prior to this project, the only practical way to author instruction was with the Linux program VIVIDS. (However, a group of developers associated with the Air Force Research Laboratory successfully authored substantial iRides instruction applications directly in the LML language, using modified Emacs editors.) Authors can also specify multipart questions in the form of tables, as shown in Figure 7and Figure 8, above. Selecting a cell in such a table opens an interface appropriate for that type of sub-question.

There is an *instruction tracing* interface in iRides that lets users see the currently active instructional elements in an active lesson plan. Authors can troubleshoot lessons by taking the role of a student interacting with the lessons and observing the flow of instructional activity in the instruction tracing view. Commands are available for pausing and resuming instruction.

This view of a lesson can be opened by using the Lesson Dataview command on the View menu in iRides Author.

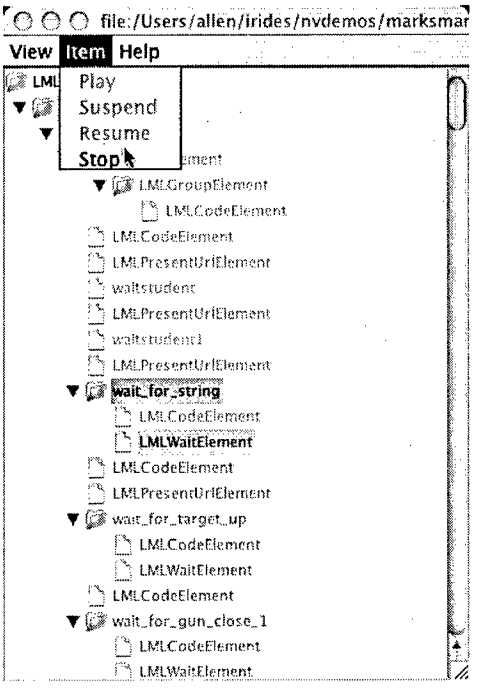


Figure 9. Tracing Instruction

2.4.5. *Authoring with higher-level instructional templates.* Authors can create reusable instructional templates (not to be confused with the simulation object templates mentioned above). These 'micro-lessons' written in LML can be invoked from calling lessons, which can specify the content for each particular invocation of a micro-lesson. Furthermore, one instructional template can invoke another, passing on the content with to its subordinate template. This facility makes it possible to build up larger instructional patterns from smaller ones.

The iRides Author package has been designed to support extensions to the instructional authoring system. A programmer can create a new template-content specification dialog, which is used to fill out the content choices for a particular use of an instructional template. IRides Author has a number of templates with corresponding authoring interfaces. These templates provide the core functionality of the *scoring items* of Classic VIVIDS. The meanings of these basic templates can be summarized as

Require that a student point to an object
    Remediate if the student fails
Watch for a particular simulation state and respond
Require that a student read an indicator
Require that a student carry out an action
    Remediate if the student fails to carry out the action
Require that the student answer a question

Each of these templates replaces a page or more of detailed LML, so the lessons that make use of these templates are much more compact and readable than if they contained all that low-level LML code at every invocation.

The extensible lesson-authoring interface has an option for authoring 'cycle' patterns of instruction. This interface is used to apply the strategy of teaching a set of items by randomly selecting from the set repeatedly. Items that are correctly responded to for a specified number of trials are dropped from the practice set. In this way, the student continues to receive practice only on the less well-learned items. Using the interface, the author may specify the initial text, the total completed answers required (defaults to the number of Query items in the cycle), the number of consecutive times a question must be answered (defaults to '1'), and the Query items that make up the cycle item.

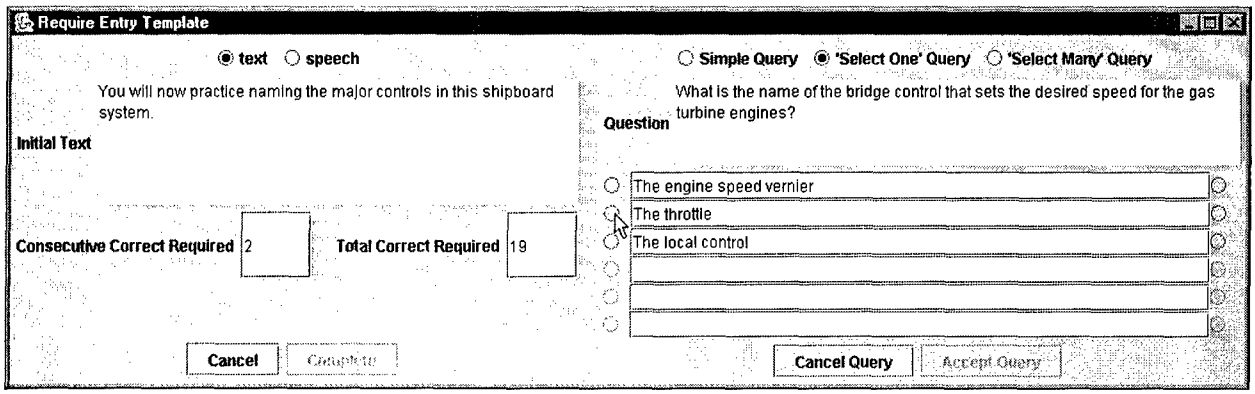


Figure 10. Authoring Cyclic Practice

3.0 Examples of simulation-based distance learning

The iRides technologies have been applied to a number of small test cases and to two major new Navy-related training areas. The first of these is conceptual aspects of Marine rifle marksmanship. The second is training on complex decision making for Navy engineering duty officers.

3.1 *Conceptual Aspects of Marine Marksmanship—Databook Training*

The Databook Training research product provides an interactive graphical environment for Marines to refresh their knowledge about the correct use of the data book. It consists of four modules designed to teach Marines concepts and conceptual skills that are correlated with good performance, particularly among novice shooters. One goal was to contribute to the elimination of 'Unqualified' shooting results during annual marksmanship certification.

The databook training product consists of four components.

Sight picture—The meaning of the term; what shooters should focus on
Databook—An introduction to the correct use of the databook on the firing line
BZO—How to carry out the Battle Sight Zero procedure using the databook
Shot Group Analysis—Recognizing shooting errors from shot patterns

This section briefly describes the four modules created.

3.1.1. *Sight Picture.* The training begins by briefly pointing out that the human eye can focus sharply on objects at only one distance at a time. An animation shows the target, sight post, and rear sight going in and out of focus. The trainee clicks to continue. The appearance of each object in focus is presented again, one at a time. The trainee clicks to continue.

Then the trainee is given the opportunity to control which element is in focus. See Figure 11, below. The trainee clicks to continue. The trainee then uses the same control interface to select the correct focus (the front sight post) for shooting. If the trainee makes an error, he or she is immediately informed and is given another opportunity. When the trainee selects the correct focus point, the lesson says that it is the correct one and announces that the lesson is over.

The GUI presents overlaid images of a target, a sight post, and a rear sight. The edges of these images are 'fuzzed out' to represent the image appearance when not in focus. Trainees are able to observe the differing appearance of these images when the focus is on each of the three objects in view. The marine's understanding of proper sight picture is tested by providing the marine with a control for setting the appropriate focus for accurate shooting. See Figure 11.

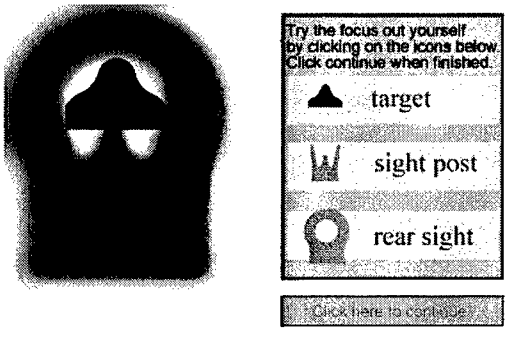


Figure 11. Sight Picture Training

3.1.2. *Databook Introduction.* This web-deliverable interactive training module presents a structured walkthrough of the use of the data book for Marines.

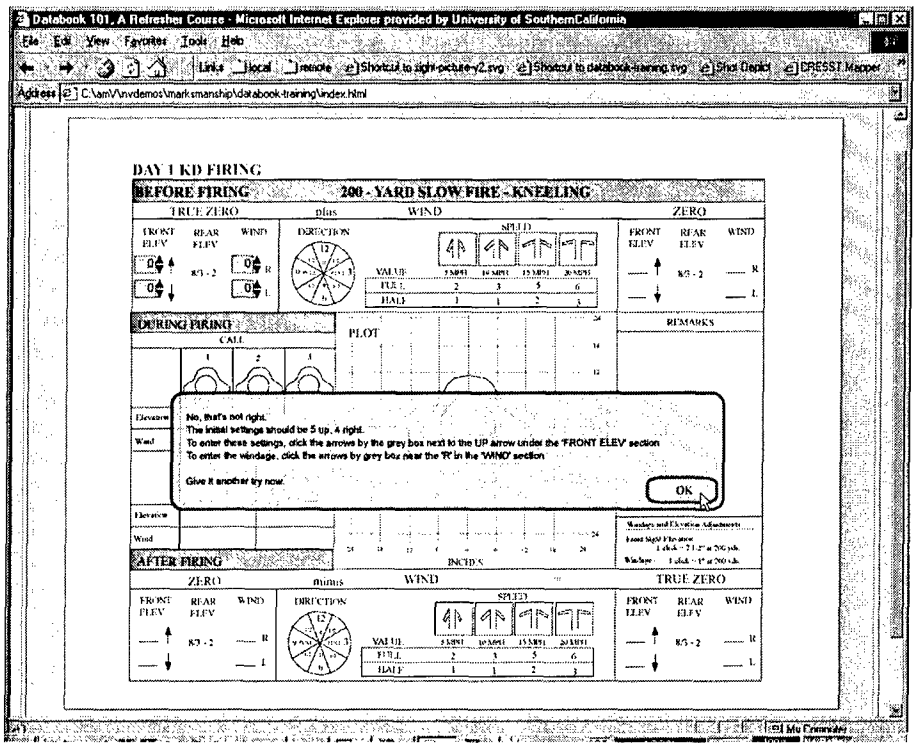


Figure 12. Review of the Meaning of True Zero.

It begins by reviewing the meaning of "True Zero" and guiding the Marine to enter the weapon's current sight settings in the True Zero section in the upper left corner of the data book page. If the student makes an error, the module points out the error and guides the Marine to enter the correct values.

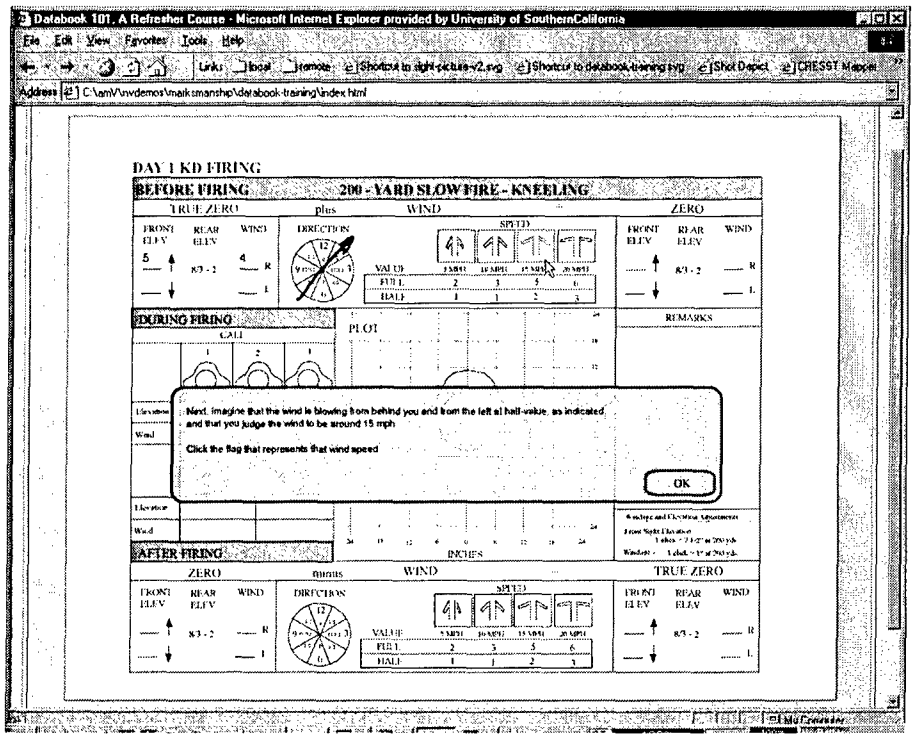


Figure 13. Zero Reflects Current Wind Conditions.

The training module then guides the Marine to consider current wind conditions, and to correctly estimate the sight adjustment that those conditions call for. It then guides the completion of the Zero section of the data book page in the upper right corner.

Once these initial settings have been correctly recorded, the Marine on the firing line will fire a group of shots and make any sight adjustments necessary. The Marine is taught that, although the actual sight settings of the weapon are recorded in the original True Zero and subsequent Zero boxes of the data book page, the During Firing section of the page is used to record adjustments, not absolute sight values.

As the training module progresses, the learner is shown the target in a sight picture context that reflects the focus of the shooter's eye as he or she estimates the point of aim at the moment that a shot is fired. The Marine must remember this point of aim so that it can be recorded in the During Firing section of the data book. See Figure 14.

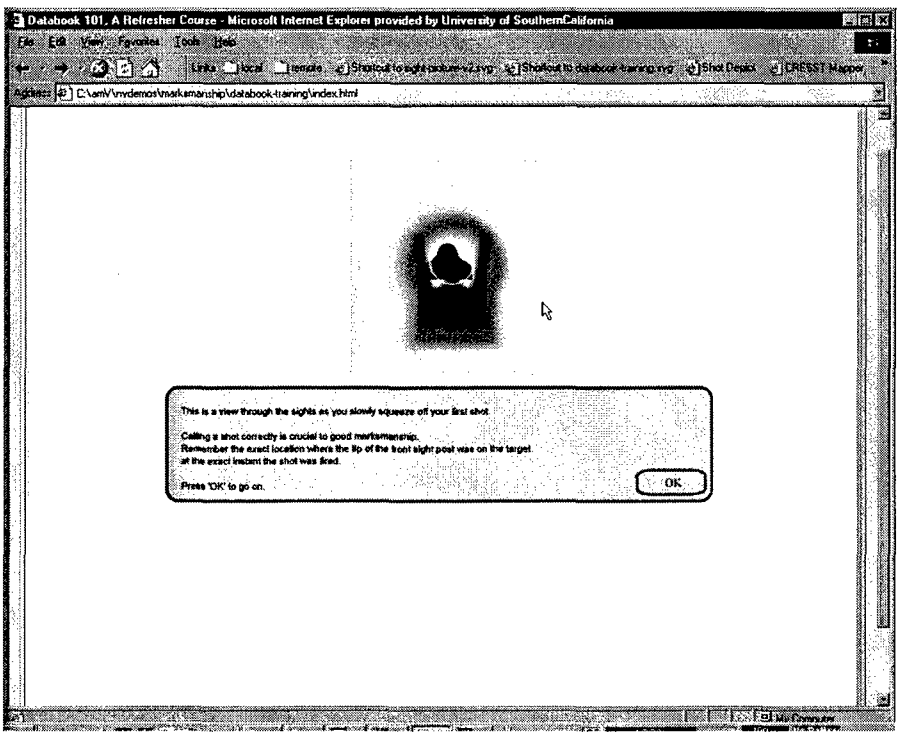


Figure 14. Sight Picture and Calling the Shot.

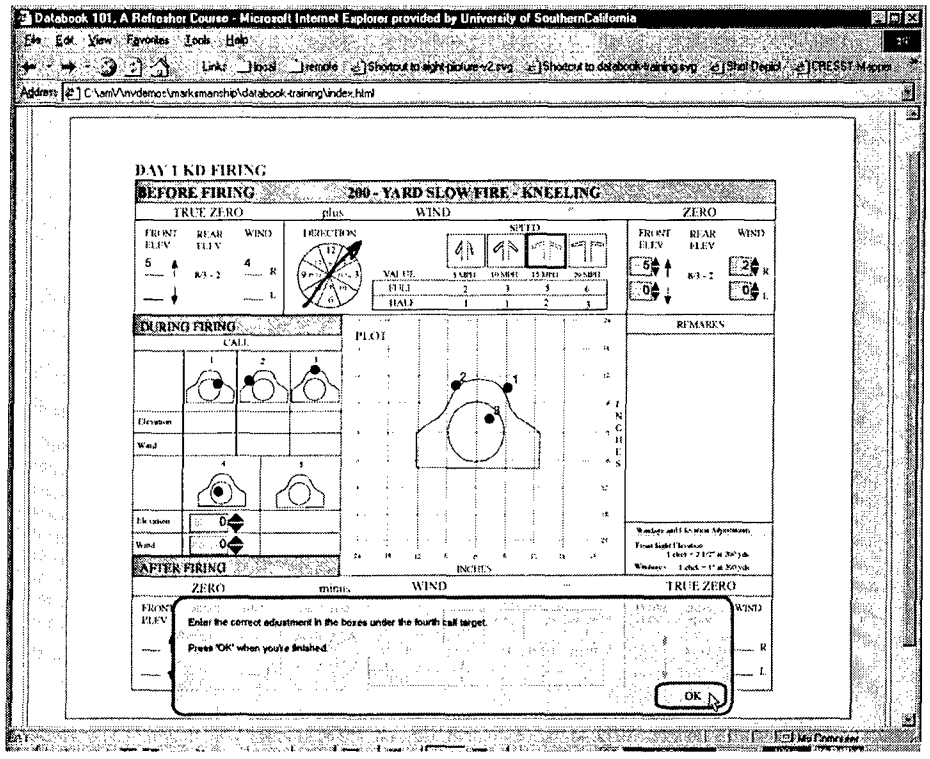


Figure 15. Adjusting Sights After the Fourth Shot.

The learner is also reminded that actual shot positions are to be recorded in the Plot section of the data book page, in the center area. After a group of three shots has been fired, the Marine is guided to enter any necessary adjustments at the fourth shot portion of the During Firing section. If incorrect values are entered, the Marine is given guidance on how to determine what values should be entered in this section.

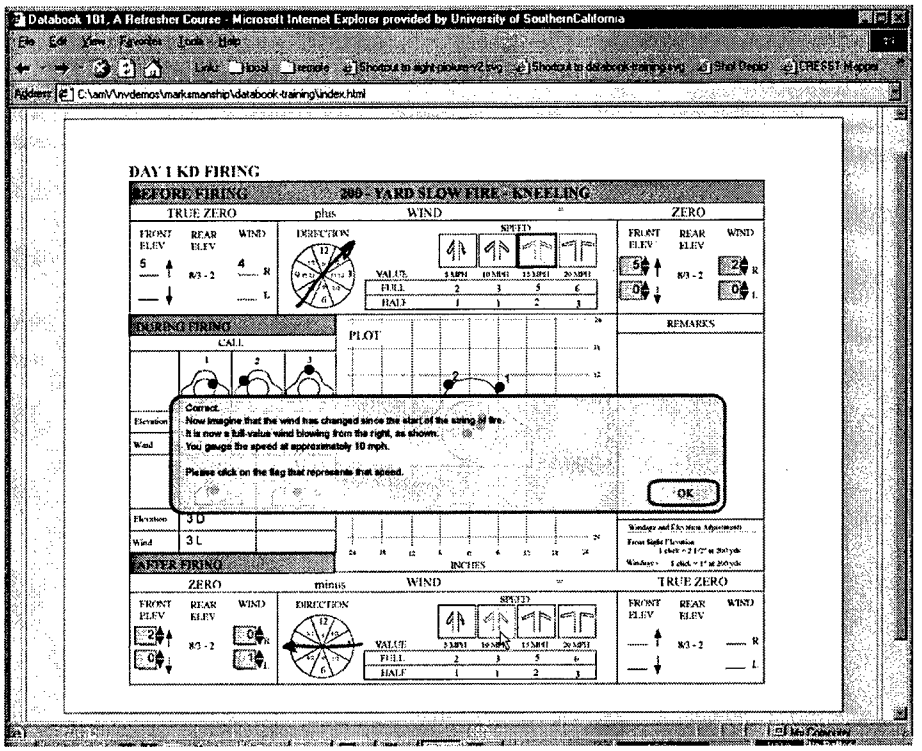


Figure 16. Deducting Windage to Record True Zero.

When the entire five-shot group has been completed, the weapon's current sight settings are to be entered into the After Firing Zero section in the lower left corner of the page. Finally, the learner is told what the final wind condition is, and is led to deduct the windage value from the Zero settings to determine True Zero.

3.1.3. BZO. The training begins by displaying a depiction of the firing range and the rifle's sights. A brief presentation on the mechanics of the simulation is made, followed by the doctrinal definition of a BZO (Figure 17).

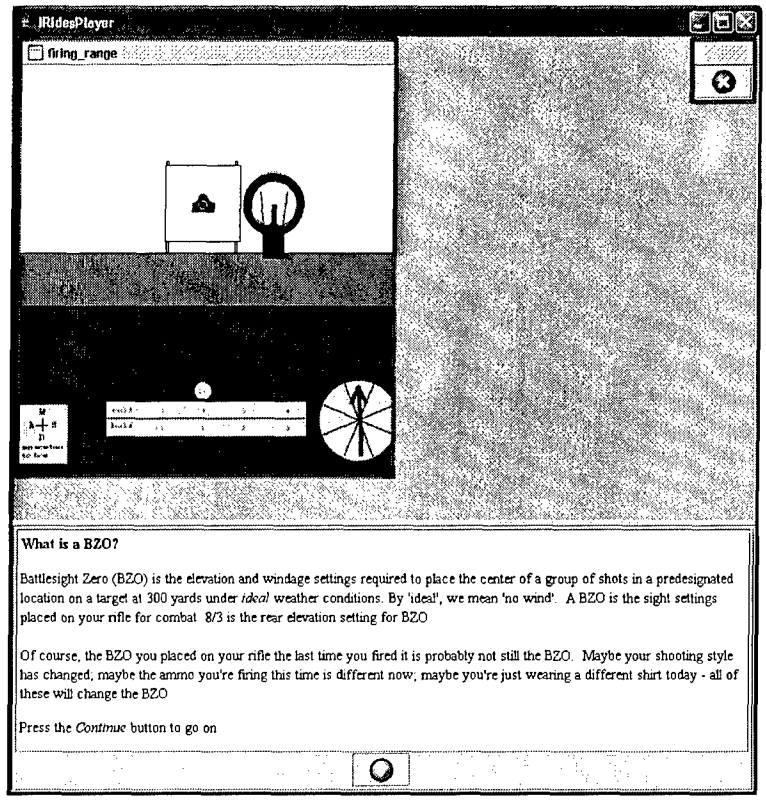


Figure 17. BZO Doctrine Presentation

The trainee is asked to take three shots at the target. If the sights are not at least reasonably close to the target, the trainee is asked to pay closer attention. After a group of shots is fired, the target is animated to mimic being pulled into the pits and being marked. When the target comes back up, the scene zooms in on the marked target to show the group, and the trainee is asked to indicate the proper sight adjustments that should be made (Figure 18), and receives feedback on the accuracy of their judgment (Figure 19).

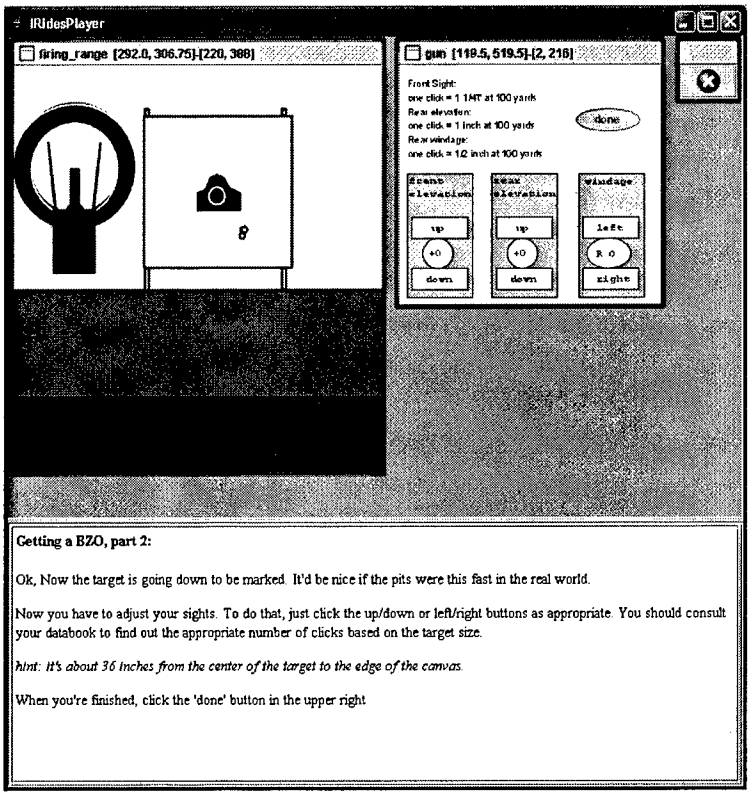


Figure 18. Creating the First Shot Group

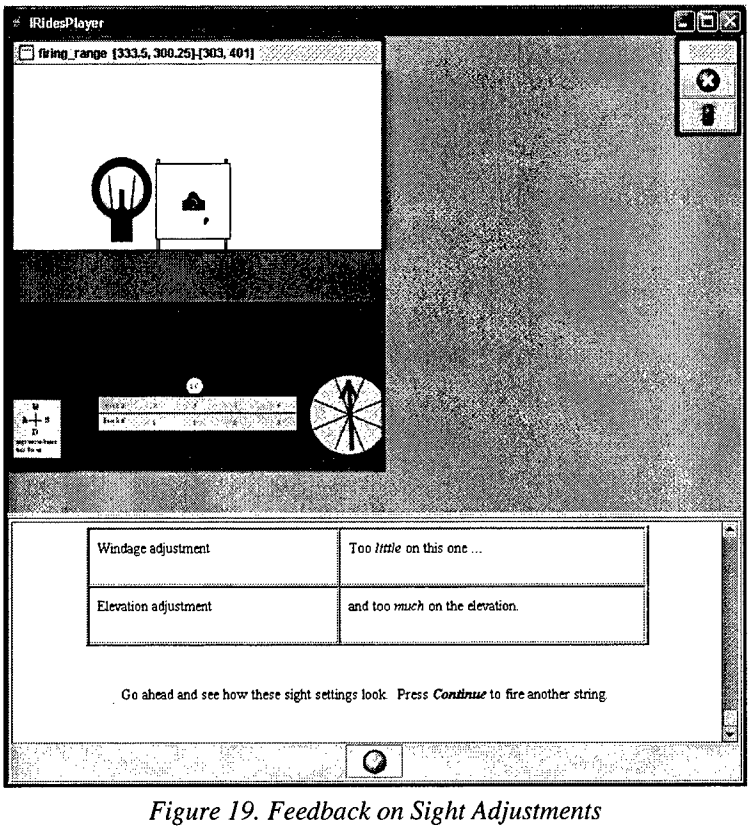


Figure 19. Feedback on Sight Adjustments

This procedure is repeated twice more, in accordance with the proper procedure for obtaining a BZO. As a conclusion, the trainee is asked to remove the appropriate windage settings in order to obtain a True Zero, and is remediated until he or she is successful (Figure 20).

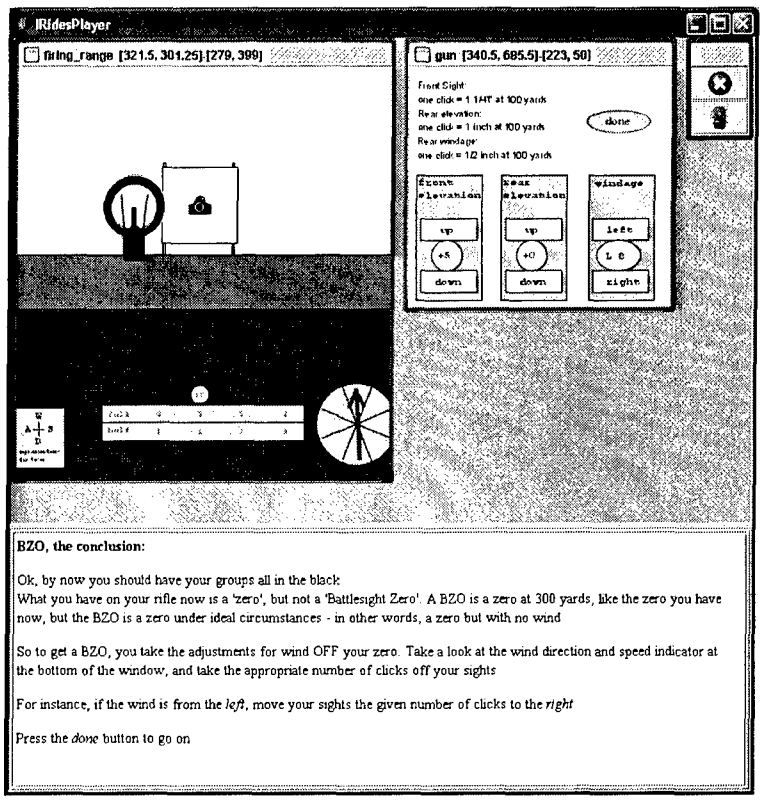


Figure 20. Explanation of Final Adjustments

3.1.4. Shot Group Analysis. The training begins with a page of the databook displayed and an 'instruction menu' from which the trainee can choose an entry point – either a discussion of shot pattern analysis, or practice at recognizing shot pattern. (Figure 21).

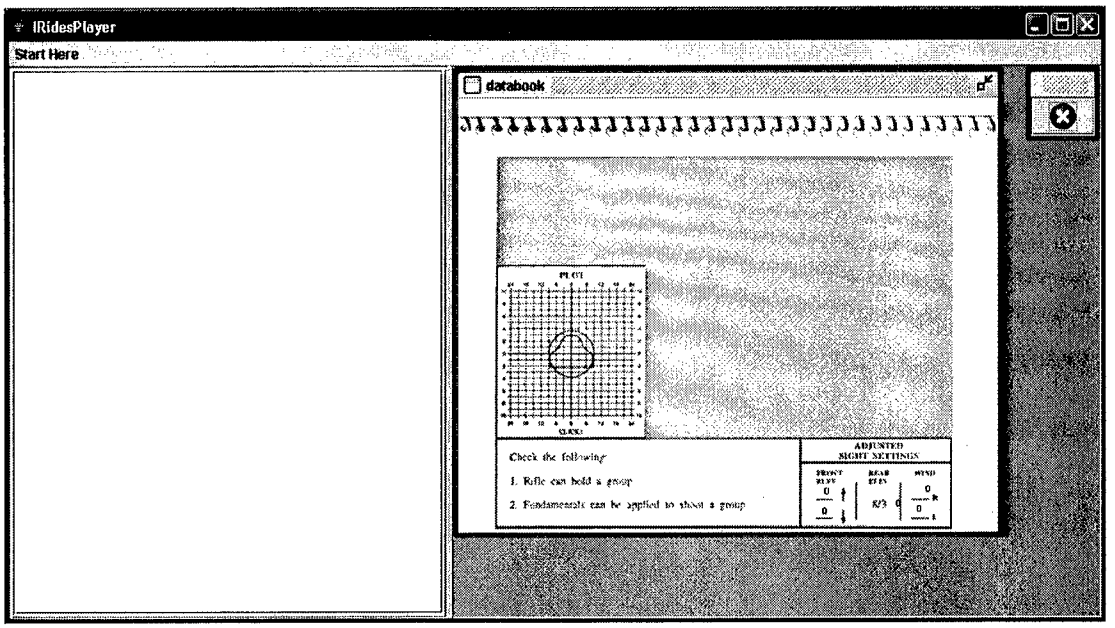


Figure 21. The Databook view in the Shot Pattern Analysis Training Module

In the 'Practice' mode, trainees are shown a series of shot patterns and asked to identify them. They are presented with questions and menus of possible answers. (See the upper left area of Figure 22.)

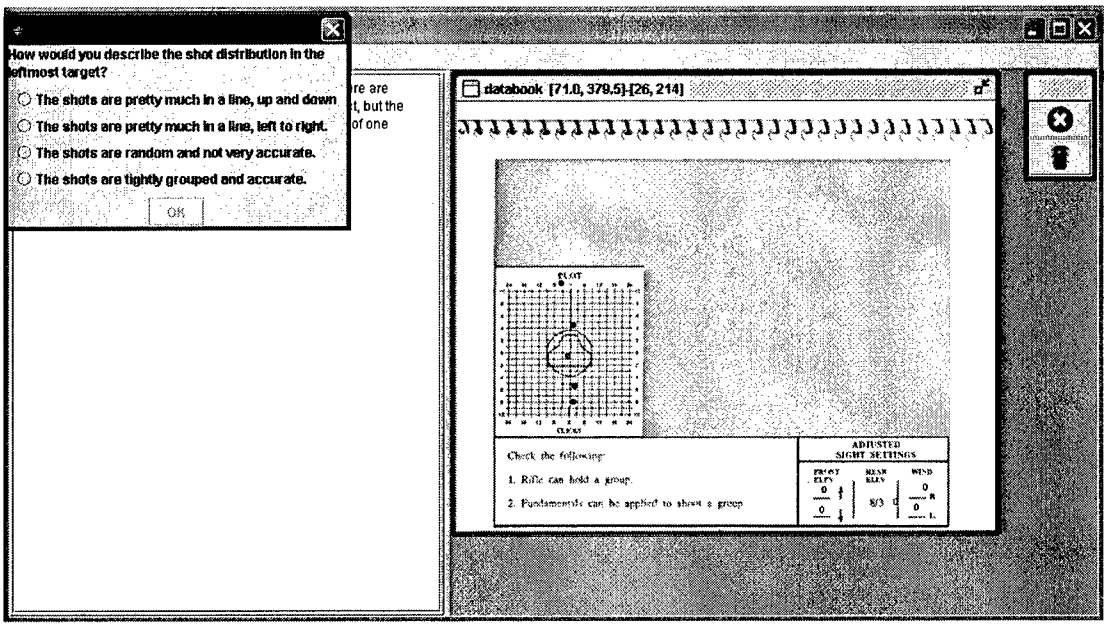


Figure 22. Question and Answer Menu

If the trainee makes an error, he or she is shown a simulation of the actual cause of the shot pattern and how it arises. Figure 23 shows the remediation for failing to recognize poor trigger control:

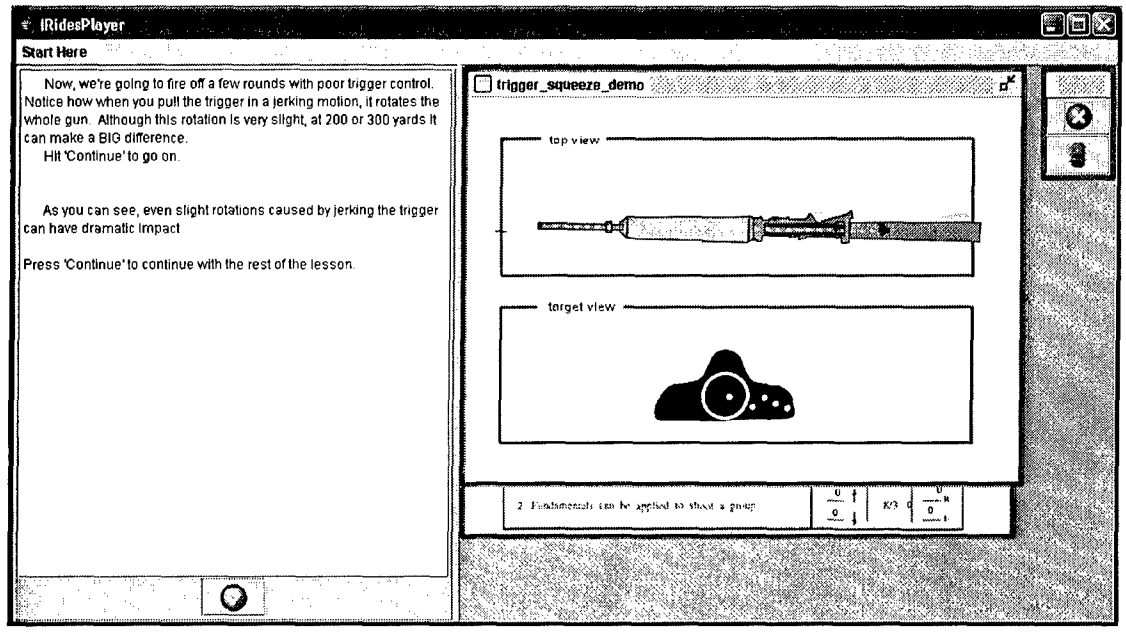


Figure 23. Explaining the Effects of Poor Trigger Control

3.1.5 Follow-on Work on the Databook. Working together, the Marine College of Continuing Education (CCE) and the Weapons Training Battalion have funded and are now managing the transition of this training software for initial use in a Rifle Coaches Training Course. From the Marine *Project Charter: Rifle Marksmanship Coaches Toolset*—

> The Marine Corps College of Continuing Education (CCE) mission is to design, develop, deliver, and manage DL products and programs for the Marine Corps, in order to increase operational readiness. The CCE's Courseware Development Department provides project management for the acquisition of DL courseware, job aids, and software tools.
>
> Weapons and Training Battalion (WTBN) Quantico is the single Marine Corps point of contact for marksmanship doctrine. The WTBN staff have been involved in on-going marksmanship research to evaluate the effectiveness of new techniques for assessing a Marines knowledge of the fundamentals of marksmanship and its impact on their shooting performance. WTBN has contributed to the development of prototype assessment instruments and training tools that positively impact shooting performance. These tools are based on the Marine Corps-approved rifle marksmanship doctrine and programs of instruction.
>
> ....
>
> - Electronic Rifle Marksmanship Data Book Training Module. The Data Book module is a software application that will train four basic marksmanship tasks: data book usage, sight picture, battle site zero (BZO), and shot group analysis. The module will be developed in two versions as student-controlled and instructor controlled training modules.

3.2 *Decision-Making Training for Engineering Duty Officers*

3.2.1. Training a Complex Decision-Making Task. The Navy's Engineering Duty Officers (EDOs) manage large scale development and procurement processes. During their initial training in a six week *EDO Basic Course*, EDO candidates are taught about making complex decisions as part of project risk management. The students, who have higher degrees in one or more engineering disciplines, must learn to make complex decisions that incorporate the uncertainty of future events, and to convincingly present their acquisition recommendations to senior Navy officers for approval. During the *Basic Course*, students are given a variety of techniques for mitigating project risk and for making complex decisions. Exercises are conducted in which teams of students analyze risks in assigned projects and make formal presentations to boards of reviewing officers, who are, in fact, faculty members of the EDO School.

When the KMT team approached the EDO School faculty about their needs for effective training and assessment of students, the faculty members asked for help in training and assessing decision-making skills in the context of the assigned project exercises. In particular, during the final exercise, students are asked to address a mid-procurement project crisis—the vendor of an important ship system (the *Refueling at Sea* system, or *RAS*) has decided to go out of the business of providing that system. Student teams must determine and evaluate possible options and present their recommendations to the review board. What could be done to make this experience one that reinforced decision-making skills taught in the course, and how could the students' application of those skills be assessed?

Several topics presented to the students in the course drew themselves to our attention. One of these is the topic of multi-attribute measures of utility. Early in the course, students are presented with an example of choosing a restaurant for dinner. Four possible restaurants are considered, and each is given a simple numerical score on such attributes as nearness, expense, atmosphere, and food quality. The concept of weighting attribute scores differentially is also introduced, and a simple Excel worksheet for computing the 'best' restaurant outcome based on weighted attribute scores is presented. At this stage of the course, the students have been exposed to these concepts:

- Multiple components of utility (attributes)
- Weighted attribute values
- Use of computer-based tools to support decision modeling

Later during the course, the faculty briefly introduces *Expected value theory* as a more sophisticated framework for making such complex decisions. In addition to estimating an outcome value for each alternative choice (by summing the weighted attribute values of all the potential consequences of that choice), the students also make estimates of the probability of each outcome, given the preceding choice. The *expected value* of a decision is computed by summing the probability-weighted estimated outcomes of that decision. Although it would be possible to repeatedly make such estimates of probability and attribute values and to repeatedly compute expected values by hand, this task clearly

would benefit from the use of computer-based support tools. At this stage of the course, the student's have also been exposed to the concepts

- Alternative decision outcomes can be assigned estimated probabilities of occurrence
- Expected values can be computed from estimated probabilities and the sums of weighted outcome attributes
- Decisions can be made based on expected value analyses

CSE and BTL decided to build an experimental tool based on the above concepts. The tool would have two purposes: to contribute to the students' understandings of these topics, and to provide a natural, problem-centered task for collecting data for assessment. The tool would have to be simple, so that students could learn to use it very quickly. That would make it possible for them to apply the tool to the RAS problem during the last project exercise during the course. In addition, it was decided that the tool would be delivered to the school with appropriate content to facilitate its use in the course. This content would include a simple version of the tool applied to the restaurant decision example. This would make it possible to introduce a simplified form of the tool when multi-attribute utility concepts were introduced early in the course. Second, a simple example—selecting a digital camera—was developed for use in introducing the concept of expected value. Third, the RAS System decision would be implemented in the tool, so that students would be able to focus on the estimates they had to make, rather than on the mechanics of authoring every aspect of the alternatives from scratch using the tool.

3.2.2. Development of the Decision Analysis Tool. The technologies used to implement the Decision Analysis Tool (DAT) were VIVIDS (Munro and Pizzini, 1998; Munro, Surmon, Johnson, Pizzini, and Walker, 1999; Munro, 2003) and *iRides Author* (Munro, Surmon, and Pizzini, in preparation). This tool was designed to enable training developers to create interactive graphical simulations and training in the context of those simulations. The iRides program can deliver the training specifications as a Java application, or over the Web as an applet or a Web Start application. The behavior specification language of iRides is sufficiently expressive and powerful that it was possible to create implementations of a real software tool for aiding decision making using weighted attributes and expected value theory.

The tool was developed in three phases, which resulted in three releases of the DAT: prototype, version 1, and version 2. After each of the first two phases, student usage and instructor comments led to significant revisions that appeared in the subsequent release. Some of these modifications were designed to make elements of the user interface easier to learn and to use, to correct algorithmic errors, and to improve data reporting. In addition, however, a number of changes were made to the tool to bring it into compliance with the specific teachings of the EDO Basic Course. Examples of this included restricting attribute utility values to integers between 1 and 5, and including three standard attributes of outcome utility: cost, performance, and schedule.

3.2.3. Using the DAT. In this discussion, the behavior of version 2.0 of the DAT is described. The data collection took place using version 1.0, but the major differences in

2.0 are not relevant to the core issues of operation sequencing in the usage of the tool. The primary difference in version 2.0 is that users are not limited in the depth and breadth of the decision trees that can be authored. In addition, the graphical user interface of 2.0 is improved by the use of Java Swing interface objects (sliders, radio button groups, check box groups, and the like) in place of authored iRides simulation objects with similar functionality.

If an author begins to develop a decision analysis from scratch, the initial display shows only a root decision node and one simple choice branch, as in Figure 24. A popup menu can be used to select among the commands that pop up when a node is clicked. On the root node, the options are "Edit Label" and "Create Choice" Create Choice is used to add a new subtree element under the root node, another possible decision choice. Other nodes have a "Delete" option, but the root node cannot be deleted, only renamed.

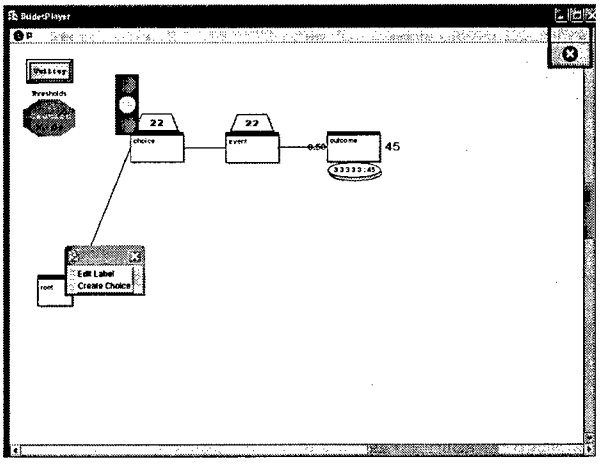


Figure 24. 'Empty' DAT Model

Authors relabel the nodes to reflect the choices in the context being analyzed and the possible outcomes of decisions. They can also create new nodes, including additional choices, events, and outcomes. At the development point shown at the right, the original nodes have been relabeled and the author has created two possible outcomes for the first evaluation: a good result, and a poor one.

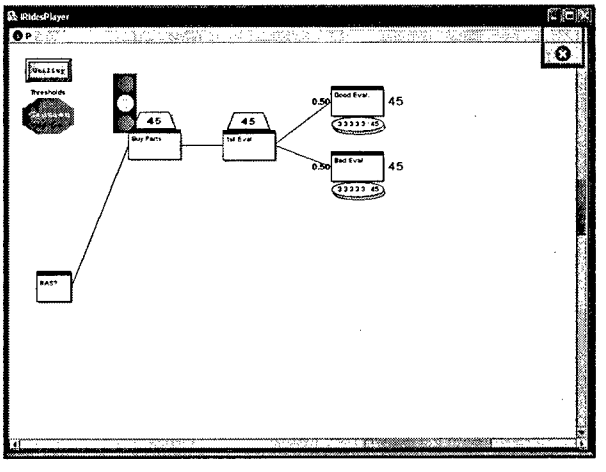


Figure 25. Renaming Nodes, Creating a New Outcome

For a given decision domain, outcomes have appropriate *attributes*. Authors can enter the names of the attributes that apply to the decision that is being analyzed. Clicking on the *Utility* button opens the Attributes Definition interface. In the original, empty DAT document, there are five attributes named "a" through "e". Each one begins with an intermediate *factor*, 3. These factors are the weights by which actual attribute values of particular outcomes are multiplied to compute the value or utility of those outcomes.

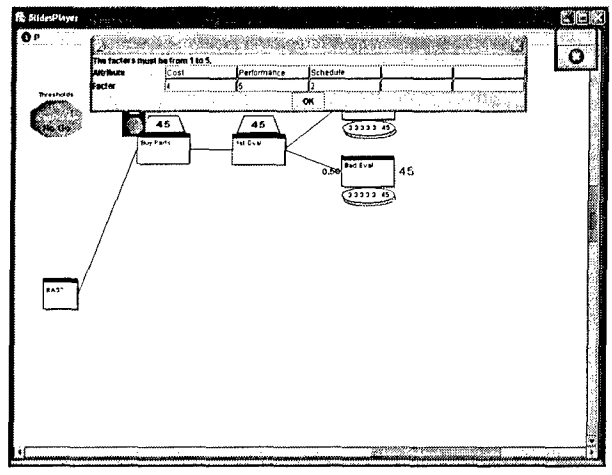


Figure 26. Defining Attributes

When the Attributes Definition interface is closed, the number of attribute values displayed below each outcome node is updated, if any attributes have been deleted or if new ones have been added. Because the original attribute names 'd' and 'e' were deleted, in Figure 27 there are only three attribute value numbers below each outcome, although there were five in the earlier figures.

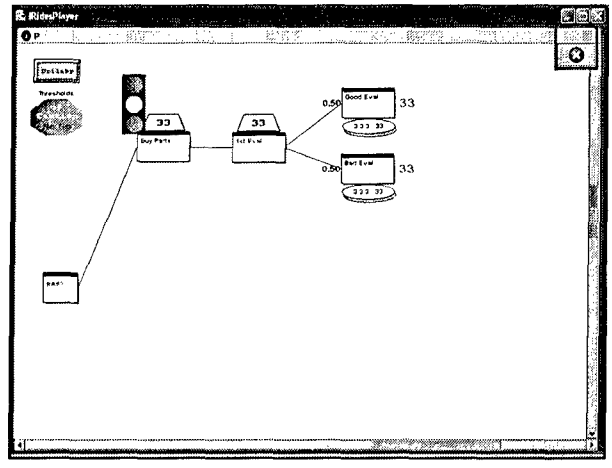


Figure 27. Updated Number of Attributes in Outcomes

Clicking on the attribute values of an outcome node opens the Attribute Settings dialog. For each outcome, the user can specify how good or bad the result will be in terms of each attribute. In the case shown here, the Cost result will be neutral (3) if the parts inventory is purchased and a good evaluation results. The performance will be excellent (5), and the Schedule will also be excellent, because roughly half of the planned production run will be completed. As these values are selected in the dialog, the numbers change in the outcome's ellipse in the main screen,

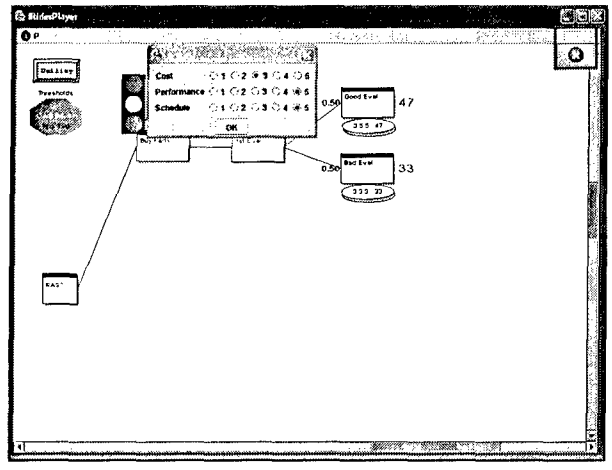


Figure 28. Specifying the Attribute Values of an Outcome

and expected values are also automatically recomputed.

Not all outcomes of a post-choice event are equally probable. The expected value of a choice is dependent not only on the utility of resulting outcomes, but also on the probability of those outcomes. Estimated probabilities are shown as numbers just to the left of outcome nodes. When new outcomes are first created, they are equally likely. (Note the 0.50 values to the left of the outcome nodes in Figure 24 through Figure 28.

Clicking on a probability opens a Probability slider. In this figure, the author has decided that there is a four percent chance of a poor first evaluation after making the "Buy Parts" decision. As the slider is dragged to a new value, the corresponding alternative outcome's probability is automatically altered so that the numbers sum to one. (If an event has three or more possible outcomes, the probability of each outcome must be set manually.)

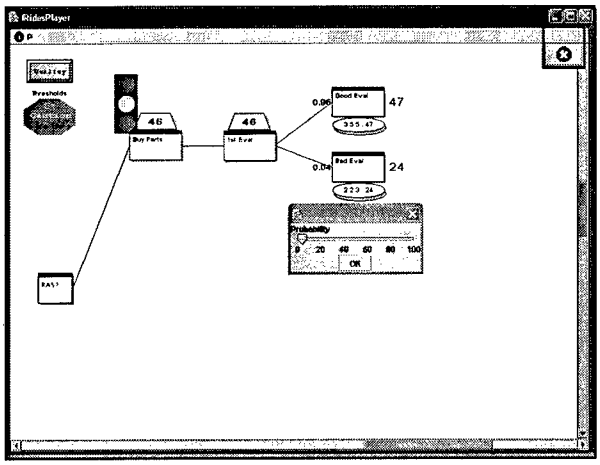


Figure 29. Assigning Probabilities to Outcomes

By continuing to add choices and outcomes, editing the attribute values, and specifying estimated probabilities, a user can develop a rich representation of many aspects of a problem. In the figure shown at the right, the values selected by the user do not result in large differences in the expected values of the choices analyzed. The 'traffic light' signal shown to the left of each choice node reflects the 'go—caution—no-go' presentation approach advocated in the EDO Basic Course. Note that all three choices are marked here with the yellow 'Caution' symbol.

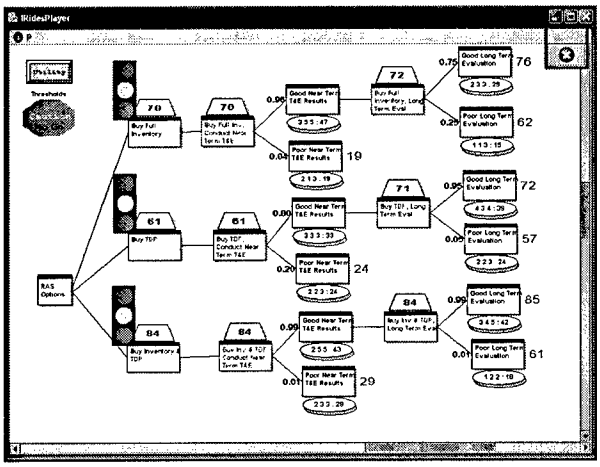


Figure 30. A Nearly Complete Decision Analysis

It is possible to manipulate the thresholds of the signals using a pair of sliders. Clicking on the button labeled "Go / Caution / No Go" pops down the slider interface, as shown in this figure. Here, the user has slightly lowered the threshold for "Go" by dragging the green/yellow slider down a bit. The third choice is now marked with a green light as the one to be preferred. Depending on how the students set the thresholds, all, some, or none of the possible courses of action they propose may produce "acceptable" outcomes.

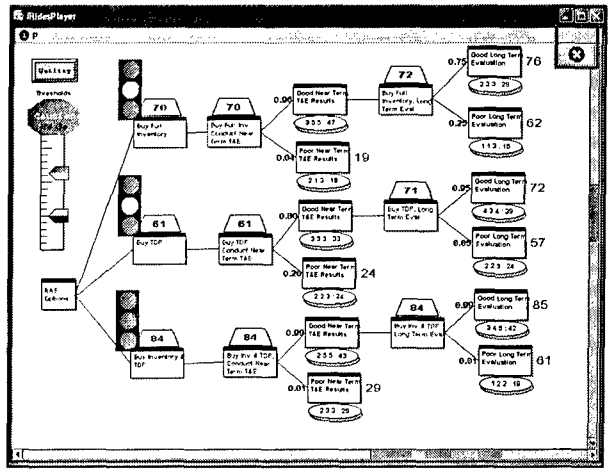


Figure 31. Adjusting Acceptance and Rejection Thresholds

3.2.4. The RAS Partial Analysis. When students begin working on the refueling at sea problem, they open a DAT model that includes three obvious choices (the three shown above in Figure 30), but with all outcomes having equal probability and equal utility. They modify these estimates to create more nearly complete analyses. They can also delete choices that they believe are not worthy of consideration, and they can devise and insert new options of their own.

3.2.5. Recording DAT Usage Data. There are seven types of events generated when students use the tool (create a new course of action, delete a course of action, label a course of action, weight the overall importance of utility attributes relative to one another, set the probability that each outcome will occur, set the utility attribute(s) of each outcome, and set the threshold values). In addition, the system will generate an eighth type of event, a "stoplight" event whenever a student action causes the expected value of a possible outcome to cross the student-determined threshold (e.g. when the value of the decision moves from acceptable to marginal or from unacceptable to acceptable).

When using the Decision Analysis Tool, each action performed by a student is recorded in an electronic "clickstream" file. Each file entry includes a student identifier, the date and time of the event, the action that generated an event and the target of the event, and the value assigned to the target of the event. For example: "EDO 33, Monday Feb 09 2004 14:45, label option 1, 'Buy Full Inventory, $20 M.' "

Because students may generate an unlimited number of procurement actions and students can evaluate an unlimited number of future decisions made about each action, we cannot identify the specific objects students will create or how they will manipulate those objects before each use of the tool. Consequently, it is difficult to evaluate a student solution by comparing it to *a single* correct solution or to compare one student solution to another student solution. Furthermore, even if we defined the largest solution space developed by any student to date as the basis of comparison, the number of degrees of freedom combined with the small number of clicks on less often chosen targets would make

meaningful parametric or non-parametric (such as Artificial Neural Network) analysis difficult. Finally, the instructors at the EDO school have indicated that they do not require such a detailed analysis.

However, as discussed above, the number of event types generated by a student remains fixed at eight regardless of the number of objects a student creates. Not only does such a reclassification have the benefit of providing a more useful level of data analysis to EDO instructors, reclassifying events at this more macroscopic level has the additional advantage of reducing the degrees of freedom to a level necessary to allow us to apply appropriate data analysis methods.

3.2.6. Instructional Guidance for the Decision Aid Tool. Initial development of the DAT has made use of the iRides simulation language and graphics, to create an interactive decision-aiding tool with data recording capabilities. The resulting product provides an environment in which concepts taught in the course could be applied in a tool context that could be quickly learned. However, the product did not make direct use of any of the iRides pedagogical features. As Self (1995) has shown, simply providing an interactive environment for experimentation is not sufficient to result in timely learning. We have since developed simple 'How to' wizards using the LML lesson specification language. Students can ask for quick reviews of basic concepts or assistance in carrying out steps in the use of the tool.

In addition, we have developed a simplified DAT analysis of the restaurants case for use in the class on multi-attribute decision theory. This will make it possible for EDO School instructors to introduce core concepts of the tool in advance of teaching about expected value theory.

4.0 Rivets—Linux-based Authoring of Simulations and Instruction

Functioning versions of *Rivets*, a descendent of the classic RIDES program have been developed for Linux and for the Macintosh under OS 10.3, with the optional X11 server installed. The application tests as a well-behaved Linux program. It can open binary files created with recent versions of RIDES and VIVIDS. Naturally, it can also save iRides simulation files (of the ".jr" type) and iRides instruction specifications, in ".lml" files. Below is a picture of a Rivets simulation authoring session taken on the Macintosh version of Rivets.

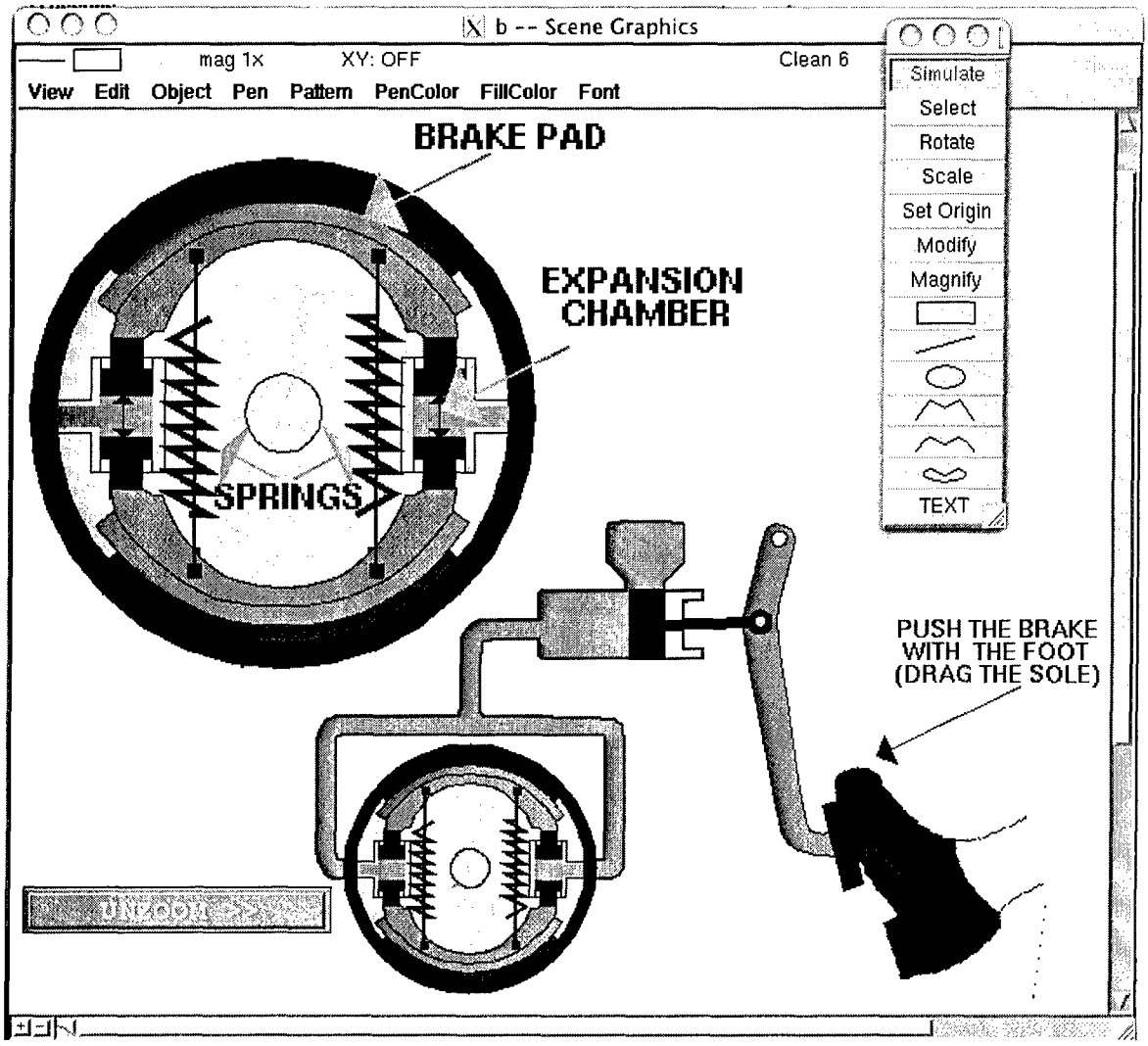


Figure 32. Macintosh Version of Rivets Application for Authoring Simulation-based Training

There are several limitations to the Macintosh version. The use of the Rivets clipboard interferes with the use of the normal Macintosh OS inter-application clipboard. The keyboard shortcuts violate Macintosh user interface standards. Other anomalies have also been identified. At this time, there are no plans to upgrade the Macintosh version. In most respects, it is as functional as the Linux version.

Having Rivets as an authoring tool option is potentially of use to authors who also have *iRides Author*. Certain features (such as object alignment options) have not yet been added to *iRides Author*. In addition, authoring certain types of procedural lessons is faster in Rivets than in iRides Author. The latter application offers more advanced simulation and instruction features, and it is a more stable program. Rivets, on the other hand, runs a bit faster (perhaps because it is written in C++ rather than Java), and it is still better-documented. Having both tools available makes it possible for an author to begin with Rivets, where appropriate, and then transition to iRides Author in order to incorporate more advanced features and to test in the iRides execution environment.

5.0 Summary

In the course of this ONR funded project, an advanced system for delivering authored interactive graphical simulations and instructional vignettes, *iRides*, was completed, tested and improved. IRides was developed with the ability to deliver simulation-based instruction in three ways: as Java applications, as Java applets, and as Java Web Start applications. The latter two options make it possible for these authored interactive graphical simulations and training to be delivered over the Web or any similar network. They thereby support advanced distance learning.

A new authoring tool, *iRides Author*, was developed for developing simulations and instruction that iRides can deliver. This tool is a Java application that provides drawing tools, behavior authoring and behavior debugging interfaces, and instruction authoring and instruction debugging interfaces. The instruction approach supports the development of novel instructional 'routines' that can be authored in XML and reused in multiple contexts. The iRides instruction authoring system can be easily extended to provide simple user interfaces that support the development of instruction using such instructional templates.

The applet version of iRides can be delivered as a SCORM compliant shareable content object (SCO). This feature is currently being exercised in a Marine Corps transition project, in which an iRides SCO collaborates with the MarineNet Learning Management System (LMS).

During the course of the development of these tools, two major efforts were undertaken to produce advanced distance learning systems, built with the tools, for Navy and Marine application. The first such effort was the marine rifle marksmanship project. Four training modules were developed for teaching or refreshing concepts about rifle marksmanship, with an emphasis on the appropriate use of the rifle marksman's Databook. These have undergone experimental evaluation in several Marine learning environments, and the Marine Corps decided to commit funds to the transition of a robust version of these research products for use in Marine classes, beginning with the Rifle Marksmanship Coaches Course.

The second major application of the iRides system was in the context of the Navy Engineering Duty Officer (EDO) School. Here iRides was used to build an application for modeling decisions where it is possible to estimate the probability and utility of the possible outcomes of a series of such decisions. This application can be used to practice this approach to complex decision-making. It can also provide instruction about its own usage and about case studies depicted in its interface. The EDO School faculty has incorporated this tool into portions of the EDO basic class.

An additional tool for authoring iRides simulations and training was produced during an extension to the grant. This tool, *Rivets*, is a fast C++ program that has been compiled for three different Unix-type operating systems: Linux, Silicon Graphics IRIX, and Macintosh OS 10.3 or later with X11. It is now possible to author iRides simulations and

training on a variety of platforms, including all the Windows platforms from Windows 98 to the present.

The flexible and open architecture of iRides makes it possible to employ this tool in collaboration with other advanced training system components. Intelligent tutors, for example, could make use of authored (maintainable) iRides simulations in advanced training systems. Additional work remains to be done to make iRides a plug-in component that can be used with a variety of advanced tutoring system approaches.

6.0 REFERENCES

Forbus, K. (1984). *An interactive laboratory for teaching control system concepts.* (Technical Report No. 5511). Cambridge, MA: Bolt Beranek and Newman Inc.

Hollan J., Hutchins E. and Weitzman L., (1984) Steamer: An Interactive Inspectable Simulation-Based Training System, *AI Magazine*, **5**(2), 15-2

Merrill, M. D., Jones, M. K., & Li, Z. (1992). Instructional transaction theory: Classes of transactions, *Educational Technology*, **32**(6), 12-26.

Munro, A. (1994). Authoring interactive graphical models. In T. de Jong, D. M. Towne, and H. Spada (Eds.), *The Use of Computer Models for Explication, Analysis and Experiential Learning*. Springer Verlag.

Munro, A. (2003). Authoring simulation-centered learning environments with Rides and Vivids. In Murray, T., Blessing, S., and Ainsworth, S. (Eds.) *Authoring Tools for Advanced Technology Learning Environments.* Dordrect, The Netherlands: Kluwer Academic Publishers.

Munro, A., Johnson, M. C., Pizzini, Q. A., Surmon, D. S., Towne, D. M. and Wogulis, J. L. (1997). Authoring Simulation-Centered Tutors with RIDES. *International Journal of Artificial Intelligence in Education*, **8**, 284-316.

Munro, A. and Pizzini, Q. A. *VIVIDS Reference Manual*, Los Angeles: Behavioral Technology Laboratories, University of Southern California, 1998.

Munro, A. Pizzini, Q. A., and Johnson, M. *The iRides Simulation Language*

Munro, A., Surmon, D., Johnson, M., Pizzini, Q., and Walker, J. An Open Architecture for Simulation-Centered Tutors. In Lajoie, S. P. and Vivet, M., *Artificial Intelligence in Education: Open Learning Environments: New Computational Technologies to Support Learning, Exploration, and Collaboration.* 1999, Amsterdam: IOS Press, 360-367.

Munro, A., Surmon, D., and Pizzini, Q. (in preparation) Teaching procedural knowledge in distance learning environments. In Perez, R. and O'Neil, H. (Eds.) *Web based learning: Theory, research, and practice.* Inglewood, N.J.: Lawrence Erlbaum Associates.

Papert, S. (1980). *Mindstorms: children computers, and powerful ideas*. New York: Basic Books.

Rieber, L. P. (1992). Computer-Based Microworlds: A Bridge between Constructivism and Direct Instruction. *Educational Technology, Research, and Development*, **40**(1), 93-106.

Self, J. (1995). Problems with unguided learning. *Proceedings of the International Conference on Computers in Education.* ICCE'95.

Shute, V.J. & Glaser, R. (1990). A large-scale evaluation of an intelligent discovery world: Smithtown. *Interactive Learning Environments*, **1**:55-77.

July 2004

Appendix A

The iRides Simulation Language:

Authored Simulations for Distance Learning

Behavioral Technology Laboratories
University of Southern California
250 No. Harbor Drive, Suite 309
Redondo Beach, CA 90277

The iRides Simulation Language:

Authored Simulations for Distance Learning

Allen Munro
Quentin A. Pizzini
Mark C. Johnson

Behavioral Technology Laboratories
University of Southern California
250 No. Harbor Drive, Suite 309
Redondo Beach, CA 90277
(310) 379-0844
http://btl.usc.edu/

Supported by Office of Naval Research Grant N00014-02-1-0179 under UCLA Sub-award No. 0070-G-CH640.

USC Principal Investigators:
Allen Munro and Quentin A. Pizzini

UCLA Principal Investigator
Eva Baker

ONR Program Management:
Ray Perez
Jan Dickieson
Wally Wullfeck

UCLA Project Manager
William Bewley

Program Development:
Mark C. Johnson
Josh Walker
Quentin A. Pizzini
David S. Surmon
Allen Munro

Information in this document is subject to change without notice and does not represent a commitment on the part of the University of Southern California.

The iRides Simulation Language: Authored Simulations for Distance Learning

Contents

Acknowledgements

Development of this technical document was supported by Office of Naval Research Grant N00014-02-1-0179 to the Center for Research on Evaluation, Standards, and Student Testing at the University of California Los Angeles, Eva Baker, Principal Investigator. Researchers at Behavioral Technology Lab, University of Southern California continued the development of iRides and wrote this documentation under support from UCLA sub-award 0070-G-CH640 to USC. ONR scientific program management for this work has been among the responsibilities of Dr. Ray Perez, Ms. Jan Dickieson, and Dr. Wallace Wulfeck.

The research product described in this document underwent initial development under the sponsorship of the Office of Naval Research under research grant N00014-98-1-0510. Dr. Susan Chipman managed the ONR scientific program for that project.

Donna Darling edited the text and prepared the index for this document.

1

iRides—Delivering Simulation-based Distance Learning

Ordinary computer-based learning and computer-based training systems commonly have a *page* orientation to instructional presentation. A learner views a page of text and/or graphics, answers questions, and is then presented with another page. In a simulation-based learning system, in contrast, a learner views an interactive system that represents many of the behaviors of the domain of knowledge that the learner wants to become familiar with. In addition, the learner may also view or interact with other interfaces, interfaces that present information about the simulated system, that direct the learner to carry out actions in the simulation context, and that offer guidance.

A live simulation permits arbitrary sequences of actions by the user. In moderately complex simulations, there may be billions of possible action sequences that users can follow in using the simulation. Each action that a user takes must result in all the changes that would be observed in a real device. Consider the aircraft AC Electric Power control panel shown at the right. This panel is used to control AC power systems on certain naval aircraft.

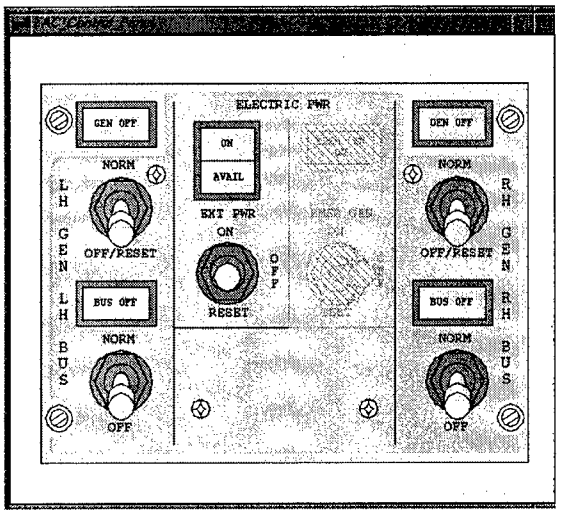


Figure 1. A Control Panel Simulation

The iRides Java based simulation engine provides a powerful and extensible simulation system and executes a language that utilizes both one-way constraints and conventional procedural constructs. Constraints are pieces of code that are executed when values they refer to change. A typical example of a one-way constraint language in common use today is found in the spreadsheet language Excel. There, a cell's value may depend on an expression containing other cell values, and will automatically be recomputed if any of those attributes change. In two-way constraint languages, such as Prolog-III or OZ, constraints are bi-directional: A change to the left side of an equality will cause the right side to adjust; while a change to the right side will cause the left side to be recomputed.

Java was chosen as the iRides implementation language because it is particularly appropriate for supporting Internet and intranet delivery of powerful software systems. (Java is **not** the language in which simulation authors develop particular simulations, however. That language, the iRides simulation language, is related to C and Java in its expression syntax. The language is interpreted by the iRides simulation parser, which is described in some detail in chapters 2 and 4 of this document.) The selection of Java for implementing iRides has supported the delivery of distance learning based on authored simulations and authored lessons. It has also made it possible for iRides to be delivered on multiple platforms. In addition to working on all Windows operating systems from Windows 95 to Windows XP, iRides also works on Linux systems and under Macintosh OS X.

The focus of this document is the iRides simulation language, not the tutorial specification and control language of iRides, LML. Some of the concepts behind the iRides tutorial system and LML (Lesson Markup Language) are described in these on-line papers:

http://btl.usc.edu/newVivids/Carroll/index.html
http://btl.usc.edu/newVivids/OpenArc/OpenArc.html

A complete and up-to-date document on the LML approach to training and tutoring will be developed in late 2004.

One of the actions that an iRides tutorial typically specifies is the loading of one or more authored interactive graphical simulations. (However, only one simulation at a time may be active in iRides.) Simulation data in iRides is stored in two parts. One part prescribes the appearance of the graphics, another part prescribes the interactive behavior of the simulation. One reason for separating the two types of simulation data is that this design will support the use of the same simulation engine with different types of graphics. The current implementation of iRides makes use of a 2D graphical specification that can be interpreted by the iRides 2D Model View component. A future iRides implementation may support a 3D graphics specification format, yet it will make use of the same behavioral specification format and the same simulation engine. Our discussion of the iRides implementation begins with a description of the simulation engine, with an emphasis on some of the innovative characteristics that support a user-extensible simulation language.

Cost Effective Simulation Development

Most action centered training systems that have incorporated simulations have relied on low level tools such as programming languages to develop both the tutors (or coaches) and the simulations. Reliance on such low-level development techniques naturally makes simulation development extremely expensive. An alternative approach is to use an authoring system to produce simulations. If all the simulations developed with a specialized authoring system automatically provide certain services to the coaching component in an action centered learning environment, then those services will not need to be re-implemented for each simulation developed.

There are a number of advantages to using an authoring system for producing simulations for use with action centered coaching systems.

- Productivity is much greater than is the case with conventional programming.
- The maintenance of complex graphical simulations is improved.
- The simulation *player* that delivers the authored simulation also provides services to the tutor. It is not necessary to code these services anew for each new simulation.

2

How the iRides Simulator Works

Constraint-based programming languages provide a number of benefits over conventional declarative languages. Foremost among these is the elimination of much of the burden on the programmer to determine the flow of control of program execution. Constraints are pieces of code that are executed when values they refer to change. In conventional programming languages, the programmer is responsible for determining the order of execution of program statements. In a pure constraint language, there is no "order" *per se*: The system of constraints represents a steady-state model of the system being simulated. When that system is perturbed by outside influences such as mouse or keyboard input, the constraint-firing mechanism in the execution environment is responsible for determining when each constraint must be evaluated and in what order. Constraint-based languages have a number of advantages, including a reduction in the complexity of the programming task, because flow of control is not ordinarily the responsibility of the constraint author. A constraint-firing mechanism in the constraint execution environment is responsible for determining when each constraint must be evaluated (executed). Several constraint-based programming languages have been developed and described (Leler, 1988; Munro, Johnson, Surmon, & Wogulis, 1993; Munro, 1994).

Constraint-based programming languages, like other programming languages, can be extended through the definition of new functions and procedures. For example, if a programming language does not have a square root function, a programmer could write a new sqrt() function in the constraint language that returns the square root of a numeric parameter. In the case of an interpreted constraint language, however, such user-defined functions typically run much more slowly than the core 'native' set of functions and procedures of the language. This is true because the body of a user- defined function will itself be interpreted, while the native functions of the language will be executed from a compiled representation.

Note: When referring to user-defined functions (which may return values) and procedures (which do not return values) this document will often use the term 'function' in preference to 'function or procedure'. The innovations described here apply equally to user defined functions and user defined procedures.

The structure of the iRides simulation language permits programmers to extend the set of 'native' functions of the iRides constraint-based language. This makes it possible to develop faster, more efficient routines. These new user functions become native parts of the language and are accessed in precisely the same way as the standard set of functions. Unlike many interpreted simulation systems, there is no execution penalty to this flexibility, since the user functions are compiled code, not interpreted code.

iRides has several techniques that together support authored simulations with the 'native' extensibility of constraint-based programming languages in the context of program execution environments, such as Java, that support reflection. These innovations are:

1 Controlling representational objects with an interpreted constraint language,
2 A method for adding 'native functions' to such an interpreted constraint language,
3 A technique for specifying the number and data types of the parameters of a user-defined constraint language's functions and procedures, and
4 A method for specifying the external triggers of user-defined constraint language functions and procedures.

Authored Constraints

Some systems for creating interactive graphical environments provide a constraint-centered approach to authoring behavior. In such systems, rather than specify what should take place when crucial events happen, an author specifies what relationships are to endure among values in a system. An example of a constraint-based environment with one-way propagation of effects is a spreadsheet authoring application, such as Excel. An author specifies that the value shown in cell D is the sum of the values in cells A, B, and C. When any of these values is changed by a user, the value of D is also changed. The author does not have to specify that the event of a value change in A or B or C should invoke the computation of D. Constraint-based languages are discussed extensively by Leler (1988). Examples of constraint-based interactive graphical environments, include Sketchpad (Sutherland, 1963), Thinglab (Borning, 1979), and a spreadsheet-based graphics system by Wilde and Lewis (1990).

In the iRides simulation system, objects have *attributes*. These attributes are used to store values associated with the objects. Relational constraints can set the values of these attributes.

An author can specify a constraint expression that determines the value of the attribute. Consider an object, a ByPassValve whose angle of rotation is constrained

to twice the rotation of another object, a SpringLever. Let this object be given the following constraint for its Rotation attribute:

```
.SpringLever.Rotation * 2
```

This rule specifies that the Rotation of the ByPassValve will be twice the Rotation of the SpringLever object.

Constraints are written as expressions. Whenever any value that is referred to in a constraint expression changes, the expression is evaluated and a new value for the attribute is determined. It is not necessary to explicitly state that the expression above must be re-evaluated whenever the Rotation of the SpringLever changes. Authors need not concern themselves with when a value needs to be recomputed if that value is determined by a constraint.

The way that representations are controlled by constraints is illustrated in Figure 2. A data stream (a file or other equivalent source of previously authored data) contains specifications for attributes, constraints, initial values, and representations (e.g., graphics). As part of the parsing process, when each representation object is created, it registers as a listener to those attributes that begin with the same name. For example, a graphic named `.Panel.ExtPwrSwitch.` would register as a listener for all the graphical attributes that can be associated with that representation. If this representation is a group graphic, then it would register for the attributes associated with group representations. In a particular implementation of the system described here these group attributes could be (for example) `Scale`, `Location`, `Visibility`, and `Rotation`. In this example, the graphic for the external power switch would register as a change listener for these attributes:

```
.Panel.ExtPwrSwitch.Scale
.Panel.ExtPwrSwitch.Location
.Panel.ExtPwrSwitch.Visibility
.Panel.ExtPwrSwitch.Rotation
```

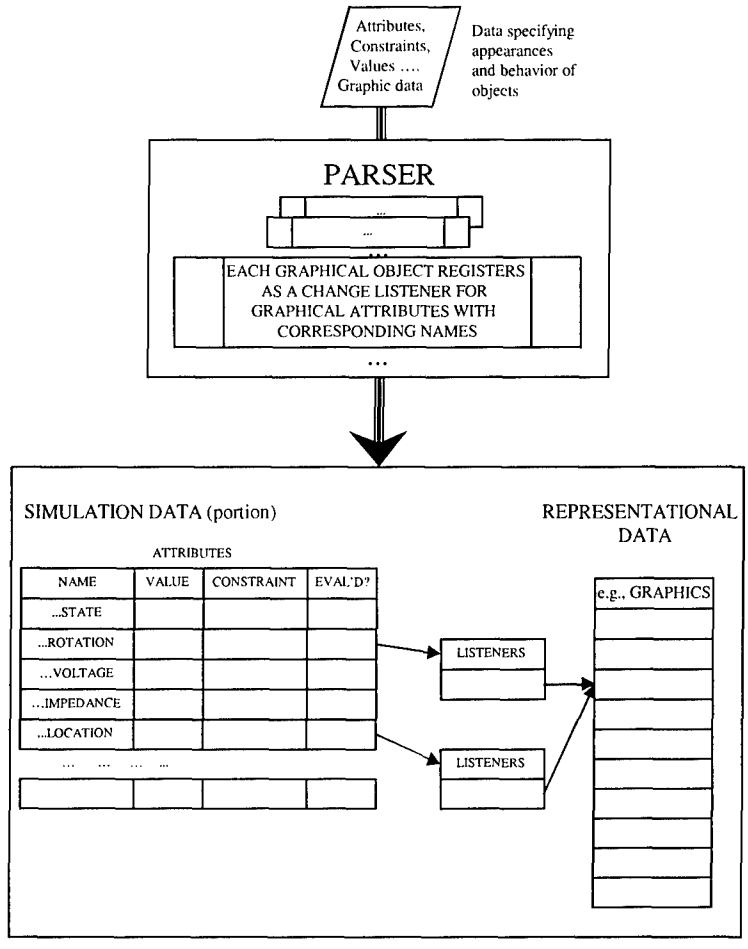


Figure 2. Representations Register as Change Listeners for Related Attributes

Whenever an attribute value changes, each listener (in the example in Figure 2, each graphic that has registered with the attribute) is informed. The listener's ValueChanged() method is called, and the new value is passed to the listener. The representation stores the new data locally, and marks itself as being in need of redrawing. The representation will mark itself as invalid. During the application's next drawing event, the representation will redraw itself, if it is then exposed to view (that is, if it is in the visible part of a window, etc.).

Propagating the Effects of a Value Change in a Network of Constraints

When the data that describes a set of constraints is parsed, the parser creates a list of dependents for each attribute. These are the constraints that refer to that attribute. Figure 3 sketches this process and the resulting simulation data. In this figure, the first attribute is referred to in the constraint of the third attribute. The fourth and fifth attributes have constraints that refer to the value of the last attribute. They are therefore dependent on that value, and they appear in that attribute's list of dependents.

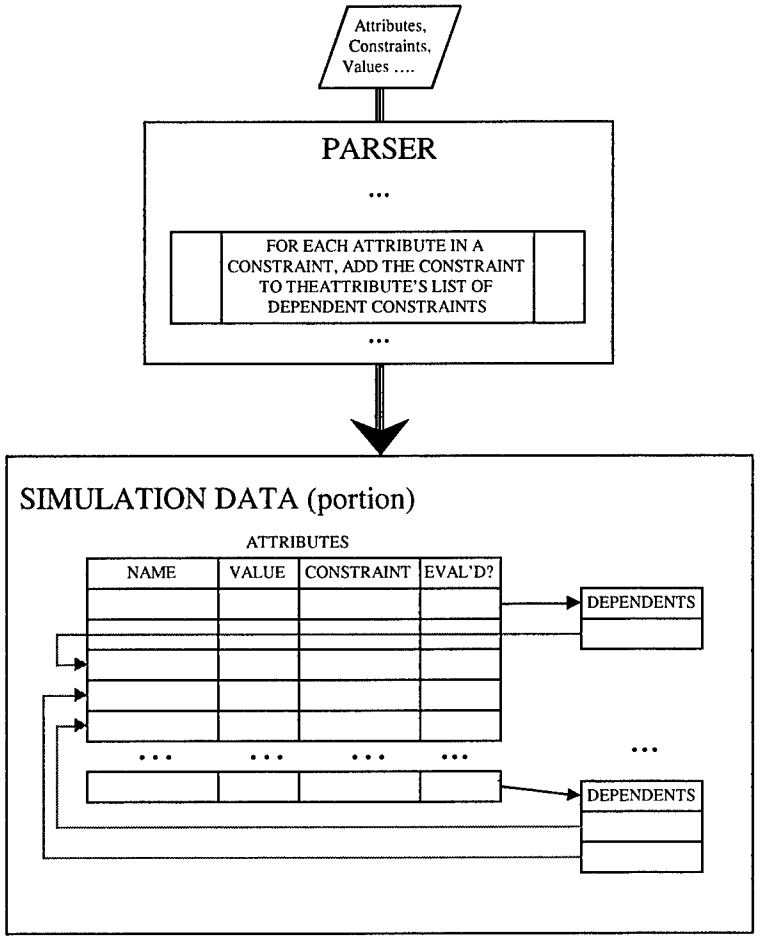


Figure 3. The Dependent Constraints of Each Attribute Are Detected During Parsing.

Constraints frequently contain references to one or more attributes. For example, in the simulation shown in Figure 1, there might be a State attribute that tracks the status of the left generator power off light (an attribute named `.Panel.LftGenPwrOffLight.State`). Its constraint expression could be

```
if .PowerSys.LeftBusPower.State="on" and .Panel.LeftGenSwitch.State="on"
   then "on"
   else "off"
```

Naturally, the exact form of the constraint language syntax is incidental to the system and methods described here. What is important is that the parser detects the names of the attributes

```
.Panel.LftGenPwrOffLight.State
.PowerSys.LeftBusPower.State
.Panel.LeftGenSwitch.State
```

and adds a reference to the first one to the dependency lists of the second and third. The attribute `.Panel.LftGenPwrOffLight.State` is *dependent* on the other two.

While the simulation is running, attribute values may change for a variety of reasons. A CurrentTime attribute, for example, might change virtually continuously. Other attribute values change when a user takes action, such as clicking a mouse in an object, or typing a value into a cell. When an attribute value does change, all the constraints that refer to that attribute will need to be evaluated. The dependents list is used to place those constraints on an activation stack. See Figure 4.

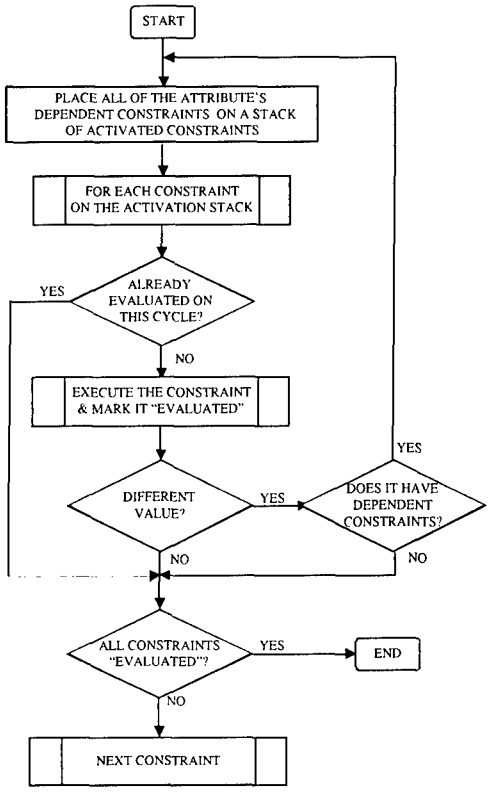


Figure 4. Propagating the Effects of an Attribute Value Change

When an attribute value changes, all of the constraints that are dependent on that attribute are placed on a stack of *activated constraints*. These are the constraints that currently need to be evaluated. For each constraint on the activation stack, the constraint execution engine must first check to see whether it has already been evaluated on this execution cycle. (Of course, the first time through, none of them will have been evaluated.) If it has been, it is skipped. Otherwise, the constraint is executed. When "native" functions are encountered in constraint executables, the *compute* method of those functions is utilized to effectuate the tasks of those functions.

The result of a constraint evaluation is examined to determine whether the value computed is different from the attribute's previous value. If not, there is no need to propagate the evaluation to *its* dependent attributes. If the new value is different, then each of the dependent constraints of this attribute must be placed on the stack of activated constraints.

This process continues until either the activation stack is empty or it has only constraints that were already evaluated on this cycle. (Temporarily exiting the constraint execution process makes it possible to carry out other tasks, such as redrawing, even when authors have written defective sets of constraints that have infinite loops.)

What this Means for the Author

Because the iRides simulation engine is managing the flow of control of rule activations, authors don't have to worry about when rules will be executed. If authors will rely on the simulator to handle such mundane tasks, their job of specifying the behavior of systems is much simplified. Authors should not misuse the Event construct in iRides to build their own simulation loops, for example. Trust the iRides simulator!

3

Simulation Language Functions

In a sense, there are two simulation languages in iRides. One is the language of events, the other the language of constraints. Most authors create simulations using primarily constraints. The event language can be seen as a superset of the constraint language—a superset that makes use of statements as well as expressions.

This chapter outlines how the statements and expressions of iRides rules are executed when the simulation is running. It then describes the meanings of the reserved words and symbols of the language.

Simulation in iRides

Simulation behavior is centered around attributes and their values. Graphical objects have intrinsic attributes whose values govern their appearance. Authored attributes may be created to compute and store intermediate results. These intermediate values may ultimately be used to determine the values of intrinsic attributes. iRides has two primary methods for specifying and changing the values of simulation attributes: constraint and events.

Constraints. Each attribute can have a unique *constraint*, an expression which specifies the value that the attribute should maintain. These are uni-directional constraints. The simulation author need only specify what value an attribute should take as a function of other attributes in the simulation. The simulator does the work of maintaining attribute values and determining the order in which affected constraints should be evaluated when the value of any attribute in the simulation changes.

Constraints are not the same as an assignment of value in a conventional programming language because a constraint does not specify when an attribute is to change value, but rather what value an attribute must have. As with spreadsheet applications and experimental constraint languages, there is a great deal of power to this approach since the author can specify simulation behavior as a set of requirements to be met rather than having to describe the steps needed to meet those requirements. However, there are cases where this approach does not provide enough power, and events must be employed.

Events. *Events* are the other mechanism by which attribute values may be changed. An event is defined to occur at an instant in simulation time and may directly affect the value of more than one attribute. Events are intended to address two fundamental drawbacks of using constraints to describe simulation behavior. First, constraints specify relationships, but in some cases there should be a simulation response to a single event in time such as a student mouse click or the moment when the value of an attribute crosses a threshold. Second, an author has no control over the order of evaluation of constraints. Sometimes control over order of execution is required. For example, swapping two attributes values is procedural by nature and cannot be expressed as a constraint.

Events in iRides address these two drawbacks of constraints. An event specifies a trigger condition which determines when the event is to occur and an event body that is the procedural definition of the event's actions.

Event triggers are specified as boolean expressions. The event is triggered only when the trigger expression is evaluated and is found to be true. Any change in a value referred to in a trigger expression will cause that expression to be re-evaluated. If the trigger is found to be true, whether or not it was also true before, then the event body will be executed.[1]

> *A Note on Thresholds.* Sometimes it is desirable to have an event fire *only* when the trigger *becomes* true. For example, it may be necessary for an attribute to be assigned the *time* at which a simulated voltage dropped below some value. (If a *constraint* were used to assign the value to the **`LowVoltageTime`** attribute, the value would keep changing as the observed voltage took on a number of different below-threshold values.) The correct way to design such a simulation is to create a logical attribute (it might be called **`BelowThreshold`**) with a constraint rule that returns **`true`** when the voltage is lower than the threshold value and **`false`** otherwise. This **`BelowThreshold`** attribute can then be made the trigger for an event that sets the **`LowVoltageTime`** attribute to **`.sys.clock`**.

Event actions are treated as taking place in sequence and instantaneously. (The simulation clock is frozen during even execution, so no simulation time passes while an event is executing.) Events can be used to enforce simultaneous attribute changes. The effects of all the changes in an event body will not propagate through the affected constraint rules until the entire event body has been performed. For

[1] Earlier editions of this document incorrectly described the role of event triggers. Thanks to the late Jeff Rickel of Information Sciences Institute for pointing out the error.

example, in an event that exchanges two attribute values one would not want effects to propagate through the network of constraints before *both* attributes had taken on their new values.

Because certain student actions, such as mouse button presses and the typing of keys, are very frequently used to modify attributes, the language of constraints permits event-like responses to these actions. The functions **DownClick**, **UpClick**, and **GetKey** can be used in constraints to prescribe instantaneous effects.

Events serve the roles of functions and procedures in other programming languages. An event can return a value, and events can take parameters. They can also have local variables

Referring to Objects. In writing rules (constraint expressions, event delays, event trigger conditions, or event bodies) it is often necessary to refer to simulation objects. In order for the rule to be understood by the simulator, the objects must be unambiguously named. The author assures this by typing not just the name of the object, but the entire 'path name' of the object. A path name consists of the name of the scene the object is in followed by the names of any grouped objects that contain the object being specified in the rule. For example, a constraint expression might take the form

```
if MouseIn(.TestStand.FrontPanel.Switches.MainPowerToggle.)
   then 100
   else 0
```

In this example, as long as the mouse is held down and is in some part of the MainPowerToggle object, a value of 100 will be returned, otherwise a value of 0 will be returned. The toggle, however, is part of the group object Switches, which in turn is part of the group object FrontPanel, which is contained in the scene TestStand. In order for this path name to be correct, the scene name must be preceded by '.', and each object name must be followed by '.'.

There are times when the path name of an object does not need to be used. If a constraint expression is being written for an attribute and the object being referred to in that rule is the object that contains that attribute, the word **self** can be used, instead of the path name for that object. In this case, the above rule would be written more simply as

```
if MouseIn(self)
   then 100
   else 0
```

Similarly, a rule can refer to a subpart of the object whose attribute has the constraint.

```
if MouseIn(UpperPart.)
   then "increasing"
   else "decreasing"
```

Note that subobjects are referred to without a preceding "." or any other part of the path.

Referring to Attributes. Just as in the case of object references, for attribute references it is often necessary to specify the entire path name of the attribute. For example, the Rotation value of one object might be set by the Value attribute of another by the following expression

```
.FrontPanel.Meter.Value *10
```

In this case the Rotation attribute being referred to is that of the Meter object, which is in the FrontPanel scene. Note that attribute names are not terminated with a '.'.

There are times when the path name of an attribute does not need to be used. If the Rotation and Value attributes both belong to the same object, it is sufficient to refer simply to the Value attribute, without specifying the complete path name. In this case, the above rule would be written more simply as

```
Value * 10
```

The attributes of subobjects also can be referred to without using full path names. For example,

```
UpperPart.TimeTouched - Scene.StartTime
```

It is also possible to refer to an attribute of an instructional item. This is done using the full path name of that attribute. However, the entire name must be prefixed with 'I:' (instructional item). For example a trigger condition in an event might be

```
I:.lesson12.keyboard_maximum.Score > 0
```

The trigger condition is satisfied if the Score attribute of the instructional item keyboard_maximum of lesson12 is greater than 0.

The iRides Simulator. In order to implement the handling of events and maintenance of constraints, the iRides simulator keeps an ordered list of current constraints (and events) and executes each one in order. This list of to-be-evaluated rules is called the *current events* list. Executing one rule on the list may cause others to be added to the appropriate place in the list. This happens when the first rule assigns a new value to an attribute that is used in another rule. The rule that uses the newly changed attribute is automatically added to the current events list. Often, there are many rules that refer to a changed attribute, and they must all go on the current events list. The simulator keeps track of what order rules should go on the list according to their dependencies. When an attribute that has an associated graphic characteristic takes on a new value, then the graphics component of iRides is informed that redrawing of that object will be required.

In addition to dealing with student actions, the simulator coordinates with the instruction manager object. At the request of the instruction manager, the simulator passes information back about certain student actions, starts and stops the simulation process, and installs requested simulation states at the start of exercises.

Logical Functions

The logical functions and operators of iRides return the values **`true`** or **`false`**.

=

<logical> **==** <logical>
<number> **==** <number>
<text> **==** <text>
<array> **==** <array>
<pattern> **==** <pattern>

The = operator (the equality operator) returns **true** if the two arguments are equal and **false** otherwise. Two arrays are considered equal if the arrays are of the same length and each element in the first array is equal to the corresponding element in the second array. For example, two colors are considered equal if they have the same Red, Green, and Blue component values.

!=

<logical> **!=** <logical>
<number> **!=** <number>
<text> **!=** <text>
<array> **!=** <array>
<pattern> **!=** <pattern>

The **!=** operator (the inequality operator) returns **false** if the two arguments are equal and **true** otherwise.

<>

<logical> **<>** <logical>
<number> **<>** <number>
<text> **<>** <text>
<arrray> **<>** <array>
<pattern> **<>** <pattern>

The **<>** operator (the inequality operator) returns **false** if the two arguments are equal and **true** otherwise.

<

<number> **<** <number>
<text> < <text>

The < operator returns **true** if the number on the left is smaller than the number on the right. In the case of texts, "<" returns **true** if the text on the left would be sorted alphabetically before the string on the right.

>

<number> **>** <number>
<text> **>** <text>

The > operator returns **true** if the item on the left is larger than the number on the right. If text values are compared, the > operator returns **true** if the value on the left would be alphabetically sorted after the item on the right.

<=

<number> **<=** <number>
<text> **<=** <text>

The **<=** operator returns **true** if the number on the left is smaller than or equal to the number on the right. In the case of texts, it returns **true** if the texts are identical or if the one on the left would be alphabetically sorted before the one on the right.

>=

<number> **>=** <number>
<text> **>=** <text>

The **>=** operator returns **true** if the number on the left is greater than or equal to the number on the right. If text values are the arguments of this operator, it returns **true** if the texts are equal or if the first text follows the second in an alphabetical sort.

!

! <logical>

The **!** operator returns **true** if its argument is false and **false** if its argument is true.

and

<logical> **and** <logical>

The **and** operator returns **true** if both arguments are true and **false** otherwise. IRides uses short-circuit evaluations of conjoined arguments. If the first argument is false, iRides need not bother evaluating the second, but can simply return **false** for the conjunction.

AltIsDown

AltIsDown ()

The **AltIsDown** function returns **true** if the **Alt** key is being held down when a key is pressed.

CtrlIsDown

CtrlIsDown ()

The **CtrlIsDown** function returns **true** if the **Ctrl** key is being held down when a key is pressed.

Defined

Defined (<text>**)**

The **Defined** function returns **true** if the attribute named in its text parameter exists and has a valid value, otherwise it returns **false**.

DeleteAtt

DeleteAtt (<text>)

The **DeleteAtt** function is used to delete the attribute named in its text parameter. If the attribute name is not well-formed or the attribute does not exist, then the function returns **false**.

Call this function only within an event.

DeleteObject

DeleteObject(<text>)

The argument must be a text value that specifies a full path to an object. The function will delete the named object from the simulation. It returns **true** on success, and **false** on failure. It can fail if the text argument does not refer to an existing object, or if the object cannot be deleted, e.g. the **.sys.** scene object. Authors are strongly urged to use this function only in events.

DownClick

DownClick (<object>)

The **DownClick** function returns **true** if the user clicks the left mouse button down in the object specified in the argument. This function and its system function **DownClick** have a special status in iRides. When a call to **DownClick** is used as the logical expression in an **if...** clause, then the expression specified in the **if...** clause is carried out as though it were the body of an event. This means that, in the context of a **DownClick** function, an attribute expression can refer to its own attribute in a way that would not ordinarily work in an attribute expression. For example, in the context of a **DownClick**, an attribute's rule may refer to the attribute in a computation, as in this example:

```
if DownClick(self)
   then ClickCount + 1
   else ClickCount
```

The **DownClick** function should only appear in the top if-clause of a constraint rule. Undefined results may be obtained if **DownClick** is nested within the **then** or **else** clause of an **if...then...else...** construct.

IsNullObject

IsNullObject (<objectref>)

IsNullObject takes an object reference as parameter and returns **true** if that object is a null object reference and false otherwise. A return value of false does not necessarily mean the the object reference parameter is valid; it is only saying that it is not the null reference. This function is useful in a situation like the following. Suppose that you want to create a clone of some template, and then once you have the clone you want to change some of its values.

```
$newobj := MakeClone(….);
$newobj.Rotation := 45;
```

The problem is that the call to Make Clone might have been unsuccessful, so when you try to set the Rotation value, the simulation will throw an exception. So, before trying to set the Rotation, the author should first check to see if the clone was

successfully created. MakeClone() return the created clone if it was successful, but it return a null object reference if it was not. So the above code should be

```
$newobj := MakeClone(…);
if IsNullObj($newobj)
   Print("some error message");
Else
   $newobj.Rotation := 45;
```

IsNumber

IsNumber (<text>)

IsNumber(<text>) takes a text parameter and returns **true** if the text value of the parameter begins with a number. Examples of text strings that **IsNumber** returns **true** for include "0", "0.0123", "723396", or "1.14e9". Text parameters with such values as "a29", "Fred", and "FF" will cause **IsNumber** to return **false**. Some text values that are not normally thought of as numbers, but that do begin with numbers, such as "0.1.2" or "1A12" will return **true**. The string can begin with leading spaces before the first numerical digit in the text, and **IsNumber** will not mind. Thus **IsNumber**(" 6.3a") will return **true**.

MakeTemplate

MakeTemplate (<object>,<text>)

The **MakeTemplate** function makes a template based on the <object> parameter. The <text> parameter supplies the name for the template. If the template creation is successful, **true** is returned, otherwise **false** is returned.

MetaIsDown

MetaIsDown ()

The **MetaIsDown** function returns **true** if the **Meta** key is being held down when a key is pressed.

MiddleDownIn

MiddleDownIn (<object>)

The **MiddleDownIn** function returns **true** if the user is pressing the middle mouse button down while the mouse pointer is in the object specified in the <object> argument. The value continues to return **true** so long as the middle mouse button is held down, even if the mouse pointer moves out of the object. Unlike **MouseIn**, **MiddleDownIn** does not return **true** if a user mouses down elsewhere and then moves the mouse pointer into the object while holding the button down.

MouseDownIn

MouseDownIn (<object>)

The **MouseDownIn** function returns **true** if the user is pressing the mouse button down while the mouse pointer is in the object specified in the <object> argument. The value continues to return **true** so long as the mouse button is held down, even if the mouse pointer moves out of the object. Unlike **MouseIn**, **MouseDownIn** does

not return **true** if a user mouses down elsewhere and then moves the mouse pointer into the object while holding the button down.

MouseIn

MouseIn (<object>)

The **MouseIn** function returns **true** if the user is holding the mouse button down while the mouse pointer is in the object specified in the <object> argument, and otherwise it returns **false**. If a logical attribute has the rule expression **MouseIn(self)**, then the attribute will be true when ever the mouse is pointing in the object and the left button is pressed. It is not necessary that the mouse was pointing to the object when the button was first pressed.

MouseUpIn

MouseUpIn (<object>)

The **MouseUpIn** function returns **true** if the user releases the mouse while the mouse is in the object. It does not matter whether the down click took place within the object or somewhere else. If the mouse was not in the object upon the upclick, **false** would be returned.

NewAttribute

NewAttribute(<text>, <text>, <value>)

This function allows an author to write events that create new attributes. The function has three arguments. The first parameter must fully specify the name of the parent object. The second parameter must provide the local name of the new attribute. The third parameter, which may be of any type—logical, attribute, etc.--provides the initial value of the new attribute. The value returned by this function is **true** if the creation of the new attribute succeeds. If the first text argument (the parent's name) is a bad reference, or if the second text argument (the attribute's local name) is ill-formed—because the name has an illegal character or is already in use--, or if the named attribute already exists, then the function returns **false**.

Call this function only within an event.

NewColorAtt
NewColorAtt (<text>, <text>, <color>)

NewLogicalAtt
NewLogicalAtt(<text>, <text>, <logical>)

NewNumAtt
NewNumAtt (<text>, <text>, <number>)

NewPatternAtt
NewPatternAtt (<text>, <text>, <pattern>)

NewPointAtt
NewPointAtt (<text>, <text>, <point>)

NewTextAtt
NewTextAtt(<text>, <text>, <text>)

The previous six functions allow an author to write events that create new attributes. Each of these functions has three arguments. The first parameter must fully specify the name of the parent object. The second parameter must provide the local name of the new attribute. The third parameter, which is of a type appropriate to the kind of attribute being created, provides the initial value of the new attribute.

The generic form of these functions is
New*Type*Att (<parentText>, <attNameText>, <value>)
The logical value returned by these functions is **true** if the creation of the new attribute succeeds. If the first text argument (the parent's name) is a bad reference, or if the second text argument (the attribute's local name) is ill-formed—because the name has an illegal character or is already in use--, or if the named attribute already exists, then the function returns **false**.
Call these functions only within an event.

NewVectorAtt
NewVectorAtt(<text>, <text>, <number>,<array_of_values>)

This function allows an author to write events that create new attributes. The function has four arguments. The first parameter must fully specify the name of the parent object. The second parameter must provide the local name of the new attribute. The third parameter specifies how large the array in the attribute's value should be. The fourth parameter is an array of initial values for the attribute. The value returned by this function is **true** if the creation of the new attribute succeeds. If the first text argument (the parent's name) is a bad reference, or if the second text argument (the attribute's local name) is ill-formed—because the name has an illegal character or is already in use--, or if the named attribute already exists, then the function returns **false**. If the <number> is less than one, the function returns **false**.
Call this function only within an event.

not

not <logical>

The **not** operator returns **true** if its argument is false and **false** if its argument is **true**.

ObjectExists

ObjectExists (<text>)

ObjectExists tests that an object with a certain name is part of the simulation. The text parameter of **ObjectExists** provides the full name of an object, including its path. If the named object exists, the function **ObjectExists** returns **true**. Otherwise the function returns **false**. Note that this function is not reevaluated when a new object is created or deleted, or when object names are changed. A logical constraint rule such as

ObjectExists(myTextAttribute)

is only evaluated when the text argument (**myTextAttribute**) changes. An evaluation of **ObjectExists** can be forced by including a call to it in an event that will be executed.

odd

odd (<number>)

The **odd** function returns **true** if the number that is its argument is an odd number. If the number is even, the function returns **false**. **Odd** should only be used with integers (whole numbers). If **odd** is called on a non-integer, it returns **false**.

or

<logical> **or** <logical>

The **or** operator returns **true** if either argument is true (or both arguments are true) and **false** otherwise. iRides uses short-circuit evaluations of disjunctive arguments. If the first argument is true, iRides need not bother evaluating the second, but can simply return **true** for the evaluation.

PostURL

PostURL(<text>)

This function opens a connection to a URL. Its parameter should be the fully specified name of a URL. If opening the connection is successful, the value **true** is returned, otherwise **false** is returned.

Call this function only within an event.

RightDownIn

RightDownIn (<object>)

The **RightDownIn** function returns **true** if the user is pressing the right mouse button down while the mouse pointer is in the object specified in the <object> argument. The value continues to return **true** so long as the right mouse button is held down, even if the mouse pointer moves out of the object. Unlike **MouseIn**,

RightDownIn does not return **true** if a user mouses down elsewhere and then moves the mouse pointer into the object while holding the button down.

SaveFile

SaveFile (<text>)

The **SaveFile** function saves the current simulation data to the file specified by the parameter. That file name specified is first converted to a full path name, for example, **.\simulation2** might be converted to **C:\home\allsims\simulation2**. If the save is successful, **true** is returned, otherwise **false** is returned.
Call this function only within an event.

SetBody

SetBody(<text>,<text>)

This function replaces the text of an event's body and then accepts the new body. Its first argument should be the name of an event; the second argument should be the text of the rule body that will replace the existing body. If the named event does not exist, or if the rule body expression is erroneous, then this function will return **false**. If the body expression is successfully accepted, then **SetBody** will return **true**.
Call this function only within an event.

SetRule

SetRule(<text>,<text>)

The **SetRule** function replaces the text of a constraint rule and then accepts the rule. The first argument should be the fully specified name of an attribute; the second argument should be the text of a legal rule for that attribute. If the attribute does not exist, or if the rule is of the wrong type or is erroneous, then the function will return **false**. On success, it will return **true**.
Call this function only within an event. This function is now very rarely needed. Try to find a way to avoid 'self-modifying code'!

SetTemplateRule

SetTemplateRule(<text>,<text,<text>)

The **SetTemplateRule** function replaces the text of a template rule and then propagates that change through all existing clones of that template. Its first argument should be the fully specified name of a template. The second argument is the name of the attribute of that template whose rule is to be changed. The third argument should be the text of a legal rule for the named attribute. If the template or attribute does not exist, or if the rule is erroneous, then this function will return **false**. On success, it will return **true**.
Call this function only within an event.

SetTest

SetTest(<text>,<text>)

The **SetTest** function replaces the text of an event's trigger condition and then accepts the new event trigger expression. Its first argument should be the name of an event; the second argument should be the text of a trigger condition. If the named event does not exist, or if the delay expression is erroneous or is not a logical expression, then this function will return **false**. If the trigger condition expression is successfully accepted, then **SetTest** will return **true**.

Call this function only within an event.

ShiftIsDown

ShiftIsDown ()

The **ShiftIsDown** function returns **true** if the **Shift** key is being held down.

SoundIsPlaying

SoundIsPlaying ();

This function returns a logical value (**true** or **false**). If a sound is still being played as a result of a call to **PlaySound**, this function returns **true**. Authors can use this function to make sure that a simulation does not start playing a new sound before an old sound is finished.

TextEq

TextEq (<text>,< text>)

This function returns a logical value of **true** whenever the two text arguments are equal, ignoring case. (The standard = operator returns **true** only if the two text arguments are exactly equal.)

TextEq ("PhanTasmania", "PhantasMania") returns **true**.

TypeColor

TypeColor (<text>)

TypeLogical

TypeLogical (<text>)

TypeNum

TypeNum (<text>)

TypePattern

TypePattern (<text>

TypePoint

TypePoint (<text>))

TypeText

TypeText (<text>)

These six functions can be used to test the type of an attribute. If authors write events that change constraint rules and events, then the authors should check on the types of the attributes that are being used, so that their application won't write erroneous rules that will cause the simulator to stop. In each case, a text parameter provides the full name (including the complete path) of an attribute that should be tested. If the attribute named by the text argument to the function exists and is of the specified type, then the function returns **true**. An unknown attribute returns **false**.

UpClick

UpClick (<object>)

The **UpClick** function returns **true** if the user releases the pressed left mouse button while the mouse pointer is in the object specified. As with the **DownClick** function, it is possible to use a reference to the attribute in a computation that is triggered by an upclick event.

```
if UpClick(self)
   then UpCount + 1
   else UpCount
```

The **UpClick** function should only appear in the top if-clause of a constraint rule. Undefined results may be obtained if **UpClick** is nested within the **then** or **else** clause of an **if...then...else...** construct.

VideoIsPlaying

VideoIsPlaying ();

This function returns a logical value (**true** or **false**). If a video is still being played as a result of a call to **Play_MPEG**, this function returns **true**. Authors can use this function to make sure that a simulation does not start playing a new video before an old one is finished.

Number Functions

The number functions and operators of iRides return numerical values. Number expressions can contain both numeric attributes (e.g., **.S.Handle.Rotation**) and numeric literals like **0**, **1**, **76933.8846**, and **1.5E+13**, as well as functions that return numeric values

Predefined Numeric Constants

Several numeric constants have been predefined in iRides

[blue]

<color> **[blue]**

[green]

<color> **[green]**

[red]

<color> **[red]**

[red], **[green]**, and **[blue]** are operators for providing access to the number components of color values. For example

.PanelScn.WarningLight.FillColor[red]

returns the value of the first number component of the fill color attribute of the WarningLight object on the scene PanelScn.

[x]

<point> **[x]**

[y]

<point> **[y]**

[x] and **[y]** can be used to access the X and Y values of a two-dimensional point attribute. For example

.PanelScn.PressureIndicator.Location[y]

returns the Y value of the location of the PressureIndicator object on the scene PanelScn.

Numeric Operators and Functions

+

<number> **+** <number>

The **+** operator returns the sum of its two arguments.

-

<number> **-** <number>

The **-** operator returns the difference of its two arguments, that is, the first argument minus the second.

-

-<number>

The - operator, unary minus, returns the negative of its single argument.

*****

<number> * <number>

The * operator returns the product of its two arguments.

/

<number> / <number>

The / operator returns the quotient of the first argument divided by the second argument.

abs

abs (<number>)

The **abs** function returns the absolute value of the number that is its argument. For example

```
abs (-1.293)
```

returns

```
1.293
```

acos

acos (<number>)

This is the arccosine function. Given a number parameter, it returns a number that can be interpreted as degrees of an angle.

arctan

arctan (<number>)

This is the arctangent function. Given a number, it returns a number that can be interpreted as degrees of an angle.

arctan2

arctan2 (<numberY>, <numberX>)

This function returns the arctangent of the ratio of y/x. It returns a number that can be interpreted as degrees of an angle. One can interpret this angle as the angle of rotation from the horizontal of the line segment that passes from the point [0, 0] in the local coordinate system through the point [numberX, numberY]. **arctan2** is undefined if it is passed the values (0, 0). If this happens during a simulation, an error message appears. This function returns 0 for all values of numberX when numberY = 0.

asin

asin (<number>)

This is the arcsine function. Given a number, it returns a number that can be interpreted as degrees of an angle.

cos

cos (<number>)

This is the trigonometric cosine function. The number argument is expressed in degrees.

Day

Day (<number>)

This function converts a number that expresses a date and time in arbitrary internal clock format into a number that represents the day of the month. **Day** will have a value between 1 and 31. A common way to get the current day is to use **Day** in conjunction with the iRides function **Now()**:

```
Day(Now())
```

exp

exp (<number>)

This function returns the value of e (2.7182) raised to the power specified by the number.

Hour

Hour (<number>)

This function converts a number that expresses a date and time in arbitrary internal clock format into a number that represents the hour of the day. **Hour** will have a value between 0 and 23. 0 is the hour that begins at midnight; 13 is 1 PM. A common way to get the current hour is to use **Hour** in conjunction with the iRides function **Now()**:

```
Hour(Now())
```

Len

Len (<text>)

The **Len** function can be used to find the length of a string. It takes a string as an argument and returns the number of characters in the string.

Length

Length (<array>)

The **Length** function can be used to find the length of an array. It takes an array as argument and returns the length of that array. If the parameter passed in is not an array, the value **0** is returned.

log

log (<number>)

The **log** function is the natural logarithm. It returns the log base e (2.7182) of the number specified in the parameter.

log10

log10 (<number>)

This function returns the log base 10 of the number specified in the parameter.

max

max (<number>, <number>, <number>, ...)

This function returns the largest of the two or more numbers that are its parameters. **max** is sometimes used together with **min** to ensure that an attribute's value is within a specified range, as in this construct:

```
min(MyMaxNum (max (MyMinNum, SourceAttibute)))
```

min

min (<number>, <number>, <number>, ...)

This function returns the smallest of the two or more numbers that are its parameters. See also **max**, just above.

Minute

Minute (<number>)

This function converts a number that expresses time in arbitrary internal clock format into a number that represents the minute of the hour. **Minute** will have a value between 0 and 59. A common way to get the current minute is to use **Minute** in conjunction with the iRides function **Now()**:

```
Minute(Now())
```

mod

<number> **mod** <number>

The **mod** operator returns the remainder that results when the integer (whole number) portion of the first number is divided by the integer portion of the second number. 17 mod 5 is 2, the remainder when 17 is divided by 5. **mod** always returns an integer.

The **mod** operator is often used with the **.sys.Clock** attribute to produce numerical whole number values in a range. The expression

```
(.sys.Clock * NumsPerSec) mod (MaxCount + 1)
```

will produce **NumsPerSec** numbers every second. The numbers will go from 0 to **MaxCount** and will then start over again.

***** and **/** have higher precedence than **+** and **-**. This means that, other things being equal, ***** and **/** will be evaluated before **+** and **-** in an expression. For example

```
2 + 3 * 2
```

evaluates to 8, because the multiply operation is carried out before the addition. Parentheses can be used to enforce a preferred order of evaluation for an expression. For example,

```
(2 + 3) * 2
```

evaluates to 10.

Among the six operators presented above, this is the order of precedence

- (unary minus, highest precedence)
mod
***** and **/**
+ and **-** (addition and subtraction, lowest precedence)

Month

Month (<number>)

The **Month** function converts a number that expresses time in arbitrary internal clock format into a number that represents the month of the year. **Month** will have a value between 1 and 12. A common way to get the current month is to use **Month** in conjunction with the iRides function **Now()**:

```
Month(Now())
```

Now

Now ()

The **Now** function produces a number that expresses the total number of seconds that have elapsed since midnight GMT January 1, 1970. The accuracy value of the function depends on the accuracy of the real-time clock in the computer. The raw number returned is not very easy to use as a date and time. A readable date/time string is created by using the text function **Date** with **Now()**.

```
Date(Now())
```

which produces strings such as

```
"Wed Dec 16 18:02:16 1993"
```

One can also use the numeric time functions of the iRides language to extract the current year, month, day of the month, hour of the day, and so on. Here is a synopsis of these time functions, together with the range of values each one can produce when used with **Now()** as its argument:

Second	**0-59**	
Minute	**0-59**	
Hour	**0-23**	
Weekday	**1-7**	where 1 is Sunday
Day	**1-31**	
Month	**1-12**	
Year	**e.g., 1993**	

NumAttributes

NumAttributes (<text>)

The **NumAttributes** function returns the number of attributes of the object named by its text argument. If the object name specified is invalid, **NumAttributes** returns -1.

Call this function only within an event.

NumFields

NumFields (<text>)

This function returns the number of white-space-delimited fields in its string argument.

NumObjects

NumObjects (<text>)

The **NumObjects** function returns the number of subobjects for the given path specified by the text argument. The same function can be used to find the number of scenes in a simulation, the number of top objects on a scene, and the number of subobjects within any group object. Here are some examples:

NumObjects(".")	returns the number of scenes in the simulation
NumObjects(".Scene2.")	returns the number of top-level objects in Scene2
NumObjects(".Scene2.Bob.")	returns the number of component objects in the Bob object.

When **NumObjects** is called on an object that contains both promoted, named iRides objects and pure graphics, only the iRides objects are included in the count. If the object specified by the text parameter does not exist, **NumObjects** returns -1.

Call this function only within an event.

NumRecords

NumRecords (<text>);

The function **NumRecords** is a function of type number that returns a count of the records in the file. If the file doesn't exist or isn't open for reading, **NumRecords** will return -1.

Ord

Ord (<text>);

The function **Ord** is a function of type number that returns the ASCII decimal value of the first letter of the text. For example,

```
Ord("Now is the time.")
```

will return 78, the ASCII decimal value of 'N'.

PointToPercent

PointToPercent (<object>, <point>)

The **PointToPercent** function returns a number between 0 and 1. Given an object and a point that lies on the edge of the object, it returns a number that specifies at

what proportion of the object's edge length the point is located. Slider controls often have an output or control attribute with a value that is determined by a rule like this:

```
PointToPercent(Track., Slider.Location)
```

If the specified point does not actually lie on the edge of the specified object, then the function will find the point on the edge of the object that is closest to the point parameter.

pow

pow (<number>, <number>)

The **pow** function returns the value of the first argument raised to the power of the second argument. If the first argument is zero, the second argument must be positive. If the first argument is negative, then the second argument must be an integer. (Negative numbers can only be raised by whole numbers.) If **pow** is called with an illegal parameter value, the simulation is stopped, the offending rule data view (attribute data view or event data view) is opened, and an error dialog opens that informs the user "ERROR: parameter out of range. The simulator has been paused."

random

random (<number>)

The **random** function returns a random number between 0 and one less than the number specified in the parameter. For example, **random(100)** returns integer numbers between 0 and 99. To produce random values between 0 and 1, one could use an expression like this:

```
(random (1000)) / 1000
```

A constraint rule with this form will fire continuously. In many cases, calls to the **random** function are more appropriately found in events or event-like constraints (those that are triggered by a call to **DownClick()** or **UpClick()**, for example).

round

round (<number>)

The **round** function returns a rounded (integer, whole number) number based on the number parameter. Authors must keep in mind the difference between the **round** and **trunc** functions.

round(0.5) = 1
trunc(0.5) = 0 (indeed, **trunc(0.99999)** = 0)

Search

Search (<text>, <subtext>)

This function takes two text arguments and returns the position of the second argument in the first. The return value is a number. If the subtext is not found the function returns 0. The first position is designated to be 1. The **Search** function is case sensitive.

Second

Second (<number>)

This function converts a number that expresses time in internal clock format into a number that represents the second of the minute. **Second** will have a value between 0 and 59. A common way to get the current second is to use **Second** in conjunction with the iRides function **Now()**:

```
Second(Now())
```

sin

sin (<number>)

This trigonometric function produces the sine of the angle expressed in degrees in the <number> parameter.

sqrt

sqrt (<number>)

The **sqrt** function returns the square root of the <number> parameter.

StopTimer

```
StopTimer();
StopTimer(<number>);
```

This event statement stops a timer. A possible use for this is to measure how much time passes before something occurs. It should be used in conjunction with **StartTimer**, which is described later in this document. There are 10 different timers available, numbered from 0 to 9. You can specify which timer to use by specifying a value for <number>; if you don't specify one, the default value of 0 will be used. An example of the use of StartTimer and StopTimer in an event body is

```
Rotation := 0;
StartTimer(3);
while (Rotation < 90)
   Rotation := Rotation + 2;
Print("Rotation completed in ",StopTimer(3)," seconds.");
```

If **StartTimer** is called with some parameter, say '3', and then it is called again with the same parameter, it restarts with a time value of 0. So a subsequent call of **StopTimer** with that parameter will return the time elapsed from the more recent **StartTimer** call.

Call this function only within an event.

tan

tan (<number>)

This trigonometric function produces the tangent of the angle expressed in degrees in the <number> parameter.

ToNumber

ToNumber (<text>)

ToNumber(<text>) takes a text parameter and returns the number value that the text presents. That is, the function **ToNumber("12.1")** returns the number **12.1**. If the argument to **ToNumber** is not a number as defined by the **IsNumber** function, then the function returns the number 0. In most cases, authors should provide error handling by using the **IsNumber** function before relying on **ToNumber**. That is, authors should write a rule for a number attribute (here, one called **NumValue**) that includes a construct like this:

```
if IsNumber(.Scn.Obj.NumText)
   then ToNumber(.Scn.Obj.NumText)
   else NumValue
```

If **ToNumber** is passed a text value that begins with a number (or spaces followed by a number) but ends with non-numeric characters, **ToNumber** will convert the first part to a number and ignore the rest. For example **ToNumber(" 6.3a")** will return **6.3**.

TrackMouseAngle

TrackMouseAngle (<point>, <number>)

It is often desirable to create an iRides object that can be rotated about its origin by a user. The function **TrackMouseAngle** tracks the motion of the mouse about the origin of an object when the mouse button is pressed. The <number> attribute is used to determine what should be considered the starting rotation when the mouse goes down in the object. The most common use of **TrackMouseAngle** is to determine the **Rotation** attribute of an object with a rule like this

```
TrackMouseAngle (Location, Rotation)
```

This rule takes the old **Rotation** value as the starting point for the new angle-tracking operation.

trunc

trunc (<number>)

The **trunc** function returns a truncated (integer, whole number) number based on the <number> parameter. Unlike the **round** function, **trunc** will return only the whole number part of the <number> parameter, no matter how close to 1 the fractional part is.

Weekday

Weekday (<number>)

This function converts a number that expresses a date and time in an internal clock format into a number that represents the day of the week. **Weekday** will have a value between 1 and 7. Sunday is 1, Saturday is 7. A common way to get the current day is to use **Weekday** in conjunction with the iRides function **Now()**:

```
Weekday(Now())
```

Year

Year (<number>)

This function converts a number that expresses a date and time in an internal format into the current year. A common way to get the current year is to use **Year** in conjunction with the iRides function **Now()**:

```
Year(Now())
```

Text Functions

These functions return values of type text.

concat

concat (<text>, <text>, <text>, ...)

concat is used to combine text values into new text values. All the text parameters are combined in the order in which they appear in the parameter list to produce a new text value.

Date

Date (<number>)

This function converts a number that expresses a date in internal clock format into text data—a string of characters. A common use of this function is in conjunction with the function **Now()**. See the discussion of **Now()**, in the number functions section. The string produced depends not only on the value of the function's number parameter, but also on what time zone the computer is in.

The strings produced by **Date** are always of the same length—24 characters—and each part of the string always conveys the same type of information.

The first three characters give a day of the week ("Mon", "Tue", and so on).
Character 4 is a space.
Characters 5-7 give a month ("Jan", "Feb", ... "Dec")
Character 8 is a space.
Characters 9 and 10 give the day of the month (" 1", " 2", ... "31").
Note that for the first 9 days of the month, character 9 is a space.
Character 11 is a space.
The eight characters 12-19 convey the time in 'military' or '24-hour' time.
(for example "23:02:40")
Character 20 is a space.
Characters 21-24 give the year. The value of **Date(0)** is
"**Wed Dec 31 16:00:00 1969**". (In the Pacific Standard time zone. In England, the string would be "Thu Jan 1 00:00:00 1970".)
If the numeric argument of the **Date** function is less than 0, then the string that is returned will describe a time before the time that **Date** returns for 0.

format

format (<format_text>, <any expression>, <any expression>, ...)

format is a workhorse function for producing text values in iRides. The first parameter is a string. Each instance of the character "%" in this format string will be replaced with a text representation of the corresponding expression in the list of expressions. That is, the first "%" will be replaced with a textual representation of the second parameter (the first expression following the format text). The second "%" in the format text will be replaced by a text version of the third parameter.

Here is a simple example of the use of **format**:

```
format ("Output Voltage is %", .Scn.PwrSupply.OutputVolts)
```

If this rule is used to control a text object's TextValue, then the object might look like this:

Output Voltage is 28

Any expression type can be formatted using **format**. If a text object is given this rule for its TextValue

```
format ("Location is %;\nRotation is %;\n Color is %",
      Location, Rotation, TextColor)
```

then the screen display of the object would look like this:

Location is [171.5, 266];
Rotation is 0;
Color is [0, 0, 0]

The **format** function interprets the special character combination "\n" in a special way to produce a new line. An author can use "\n" to create multiline text items. (The value cell of an attribute with such a rule—in either the object data view or the attribute data view—will show only the first line of the text. The entire text value will be used by a text object's TextValue attribute, however.)

To print a backslash character "\" in a formatted string, one must use "\\", which will print as a single backslash. To print a "%" in a string, use "\%".

FormatNumber

FormatNumber (<text>, <text>, <logical>, <number>, <number>, <number>)

FormatNumber takes a number parameter—the last of the six parameters—and returns a text value. In this respect, it is not unlike the iRides **format** function. What is different about this function is that it has five other parameters that determine how the number will be formatted when it is converted to text.

- The first text parameter is *flag text*, a sequence of flag characters. This sequence can consist of one or more of the following three characters: "-" "+" " ". These characters can appear in any order. These are their meanings:

  - Left justify the result. If this character does not appear in the first argument, right justify the result. If "-" is not present in the flag, then the text is right justified and padded on the left with zeros or blanks.
  + Begin every result with either "-" or "+", depending on the value of the result.
    The space symbol (" ") specifies that positive values should begin

with a blank instead of a "+"; negative values will still begin with "-". if there is no space and no "+", then positive numbers will not be preceded by either a space (unless they are right justified) or by a plus sign. See the examples in the table below.

- The second parameter, *type text*, can be "e" or "E" or "f". The "f" parameter means that the conversion will be into floating point format; "e" or "E" means that the conversion is into exponential format (sometimes called 'scientific notation'). The floating point number 0.0021 appears as "2.1e-3" if the "e" exponential format is chosen. (It appears as "2.1E-3" if the "E" exponential format is chosen.) In exponential format, only one digit precedes the decimal point.

- The third parameter, the logical *pad logical*, determines whether the number will be padded with leading zeros. The value **`true`** means that as many zeros as are required to pad the text out to the number of characters specified by the width parameter.

- The fourth parameter, the number value *width*, specifies the minimum number of characters that the text is to contain.

- The fifth parameter is a number that specifies the desired *precision* of the converted number—how many characters should appear to the right of the decimal point.

- The sixth parameter is the number that is to be converted.

Any of the parameters for **`FormatNumber`** can be attribute references or any other legal expression of the appropriate type.

The table below presents a few of the possible specifications for printing numbers.

FormatNumber statement (where `TheNum` is 12.3333)	*Result:*
`FormatNumber ("-", "f", false, 8, 2, TheNum)`	`"12.33   "`
`FormatNumber ("- ", "f", false, 8, 2, TheNum)`	`" 12.33  "`
`FormatNumber ("", "f", false, 8, 2, TheNum)`	`"  12.33"`
`FormatNumber ("+", "f", false, 8, 2, TheNum)`	`" +12.33"`
`FormatNumber ("-+", "f", false, 8, 2, TheNum)`	`"+12.33  "`
`FormatNumber ("", "f", true, 8, 2, TheNum)`	`"00012.33"`
`FormatNumber (" ", "f", true, 8, 2, TheNum)`	`" 0012.33"`
`FormatNumber ("", "e", false, 9, 2, TheNum)`	`" 1.23e+01"`
`FormatNumber ("-", "E", false, 9, 2, TheNum)`	`"1.23E+01 "`

`GetField`

`GetField` (<text>, <number>)

This function returns a text value that is the *ith* space-delimited field specified by the number argument *i*. If there is an error condition (e.g., the text argument is empty, or there is no *ith* field), then the function returns "", the empty text string.

GetFullName

GetFullName(<reference>**)**

This function takes either an attribute or an object reference as an argument. **GetFullName** returns the referenced item's full path name. For example, **GetFullName(.sys.clock)** returns ".sys.clock". Warning: rules using this function will NOT automatically fire when the name of the referenced item changes.

GetKey

GetKey()

This function returns a one-character string just when a key is pressed and any scene window has the focus. This rule for the TextValue of an object will make it display what is typed from the keyboard:

```
if DownClick(self)
   then ""
   else concat (TextValue, GetKey())
```

This rule also lets the user clear the text of the object by clicking on it.

GetKey works something like **DownClick** and **UpClick** in that it has an instantaneous effect only. In many ways, a constraint rule with **GetKey** is like an event.

GetName

GetName(<reference>**)**

This function takes either an attribute or an object reference as an argument. **GetName** returns a text value with the referenced item's local name. For example, **GetName(.sys.clock)** returns "clock". Warning: rules using this function will NOT automatically fire when the name of the referenced item changes.

GetNthRecord

GetNthRecord (<filename>, <record number>**);**

This event statement returns the Nth line from a file. The file must have already been opened with the **fopen** function. If the **GetNthRecord()** function is called on a file that has not been opened, then the string `"** NOT OPEN **"` is returned. If the line does not exist in the file, "** EOF **" is returned.

GetRule

GetRule(<text>**)**

This function returns the text of a constraint rule. Its parameter should be the fully specified name of an attribute. If the parameter does not name a real attribute, the function returns the empty text value, "".

Call this function only within an event.

GetURL

GetURL(<text>)

This function returns the text of a URL. Its parameter should be the fully specified name of a URL. If the parameter does not name a real URL, the function returns the empty text value, "". If the URL does not exist, an error will result.
Call this function only within an event.

LCase

LCase (text)

This function returns a text value that is the lower case conversion of the text argument that is passed to it. The alphabetic letters in the result of a call to **LCase** are always all in lower case.

MouseIsIn

MouseIsIn()

MouseIsIn() returns a text value that is the name of the object the mouse is in while the mouse is down. If the mouse is not down then this returns the empty text value, "".

NthAttribute

NthAttribute (<text>)

This function returns the name of the Nth attribute of the object specified by the text parameter. If the object name specified by the text argument is invalid, or if the number is less than 0 or not an integer or larger than the number of attributes of the object, then the **NthAttribute** function returns the empty text value, "".
Call this function only within an event.

NthObject

NthObject (<text>)

This function returns the name of the Nth subobject of the object specified by the text parameter. If the path name specified by the text argument is invalid, or if the number is less than 0 or not an integer or larger than the number of subobjects, then the **NthObject** function returns the empty text, "".
Call this function only within an event.

ParentPath

ParentPath (<text>)

ParentPath takes a text argument that fully specifies an object or an attribute (for example, ".sys.clock" or ".sys.Fred.Head."). It returns the *path* part of the full text specification; that is, it returns all the text up to the leaf object or attribute specified. Here are some examples of calls of **ParentPath** with what they will return.

```
ParentPath(".sys.clock")                    ".sys."
ParentPath(".sys.Fred.Head")                ".sys.Fred."
ParentPath(".sys.Fred.Head.Rotation")       ".sys.Fred.Head."
ParentPath(".sys.")                         "."
```

```
ParentPath(".")                                    "."
ParentPath("Fred")                                 ""
```

Note that if the function is called with a text argument that is not a well-formed iRides name (the "Fred" example), then the **ParentPath** function will return the empty text string, "". Note, too, that **ParentPath** does not actually check that there is any such object or attribute in the simulation. All it does is strip off the last name in a well-formed fully specified iRides object or attribute *name*.

Print

Print (<any expression>);

The **Print** function will produce output to the term window. The expression is optional; if used, it can be any expression that returns a value.

Print(); will just output a blank line to the terminal.

Print(4 * 3); will output "12".

Print("Scale ",[2,2]," Rotation ",Rotation * 4); If the initial value of Rotation was 11, this will output

Scale [2,2] Rotation 44

What is especially interesting about this function, however, is the way it returns a value; it returns the value of the last expression in the parameter list. In this case it would return 44. The function can appear within an event or as an attribute relation. For example, if it was entered as the relation of some attribute, call it **NewValue**, then when **Rotation** changed, the output to the term window would occur and the value of **NewValue** would be changed to 44. This is a very useful debugging tool. It is defined in such a way that it always returns the value, but it does not always produce output. If the property **btl.sim.debuglevel**, in the file **btl.prp**, is given a value less than 15, the output will occur; if the value is 15 or more, there will be no output.

readline

readline (<filename>)

This expression will read the next line from the named file if the file is open for reading. If the file is not open for reading, then the function will return the string "** FILE NOT OPEN **". The string returned does not include the newline character found in the file. If no more lines are available in the file, then the function returns "** EOF **". A well-designed event that reads from a file should check for these values.

SubText

SubText (<text>, <start_number>, <length_number>)

SubText is used to access a portion of a text value. It returns a value that is the portion of the text parameter (the first argument of **SubText**) that begins with the <start–number> character in the original string and extends for <length_number> characters.

SubText can be used with the **Date** function to extract parts of the constant-format date string. For example,

SubText ((Date(Now()), 1, 3)
will return the first three characters of the date text, which will always be the day of the week, e.g., *Wed*.
SubText ((Date(Now()), 5, 6)
will return characters 5 through 11, which will show the month and date, e.g., *Dec 16*.

UCase

UCase (text)

This function returns a text value that is the upper case conversion of the text argument that is passed to it. The alphabetic letters in the result of **UCase** are all capitalized.

Color Functions

Each of the following functions returns a value that is an array of three numbers.

HSVtoRGB

HSVtoRGB (<hue>, <sat>, <value>)

RGBtoHSV

RGBtoHSV (<red>, <green>, <blue>)

While the RGB (Red-Green-Blue) color model is used underlyingly in iRides, it is sometimes useful to be able to express colors in terms of their hue, saturation, and value (intensity). This color model is called the HSV model. The HSV model is a more useful way of computing changes in shading (using saturation and value while holding hue constant) than is the RGB model. The functions **HSVtoRGB** and **RGBtoHSV** translate between these two color models.

MakeColor

MakeColor (<text>)

MakeColor is used to create a color value, given the name of a color. The X11 window system provides a set of standard color names, including "Dodger Blue", "Indian Red", and "Light Slate Gray" as well as such conventional colors as "Blue" and "Yellow". Which color names are supported and what those colors mean can vary among different Unix computers, depending on the X implementation and local color data.

MakeColor

MakeColor (<red_number>, <green_number>, <blue_number>)

MakeColor is used to create a color value, given numerical specifications for the red, green, and blue components of the color. Each of these numbers should be a value between 0 and 1.

If the **FillColor** of an attribute is made to depend on numbers that are continuously changing in the range of 0 to 1, then the color of the object will change continuously.

Object Reference Functions

MakeClone

MakeClone (<text>, < text >)
MakeClone (<text >, < text >, <object_name>)

This function creates a clone object. The first parameter specifies the name of the template that is to be used to create the clone. The second parameter is the name to be given to the clone. If the third parameter is included, it specifies the name of the object that is to be the parent of the new clone. If it is not specified, the parent of the object that was the source of the template will be the parent of the clone. If the clone is successfully created, **true** is returned, otherwise **false** is returned.

Point Functions

Each *Point* function returns a value that is an array of two numbers. Most of these functions can be viewed as specific to the current 2D graphics implementation of iRides. (For this reason, these functions are found in btl.sim.user rather than in btl.sim.functions. See Chapter 3 for more information on this topic.) In other implementations of the iRides architecture, different functions might be required (as in the case of 3D graphical simulations) or different implementations of these functions would be needed.

ConstrainMouseToEdge

ConstrainMouseToEdge (<object>)

ConstrainMouseToEdge is used to find the point on the edge of an object that is closest to the location of the mouse when the mouse left button is pressed. This function is often useful when one object is to move in such a way that it tracks the mouse, but is constrained to a path. The rule for the **Location** attribute of the tracking object will look something like this:

```
ConstrainMouseToEdge (.Scn.MyPathObj.)
```

This function is very useful for building sliders and trackers.

ConstrainMouseToFill

ConstrainMouseToFill (<object>)

ConstrainMouseToFill is used to find the point in an object that is closest to the location of the mouse when the mouse left button is pressed. This function is often useful when one object must move in such a way that it tracks the mouse, but is constrained to stay in its 'containing' object.

ConstrainToEdge

ConstrainToEdge (<object>, <point>)

ConstrainToEdge is used to find the point on the edge of an object that is closest to a specified point. This function is often useful when one object must move in such a way that it tracks another moving object, but is constrained to a path. The rule for the **Location** attribute of the tracking object will look something like this:

```
ConstrainToEdge (.Scn.MyPathObj., .Scn.MyMasterObj.Location)
```

As **MyMasterObj** moves around, the object with this **Location** rule will shadow the movement by moving as close as it can be to the **MasterObj** without leaving its path object's edge.

ConstrainToFill

ConstrainToFill (<object>, <point>)

ConstrainToFill is used to find the point in an object that is closest to a specified point. This function is often useful when one object is to move in such a way that it tracks another moving object, but is constrained to stay within a closed object. The rule for the **Location** attribute of the tracking object will look something like this:

```
ConstrainToFill (.Scn.MyHomeObj., .Scn.MyMasterObj.Location)
```

As **MyMasterObj** moves around, the object with this **Location** rule will shadow the movement by moving as close as it can be to the **MasterObj** without leaving its home object's interior.

All four **Constrain...** functions have an object parameter that is used to set bounds (or *constraints*) on the point value that is returned. In the case of the functions that constrain to a fill area, the point that is being returned will be a point within the fill area of the constraining object. In the case of *line* type objects, there is no fill, but the point is constrained to be on the line. In the case of *text* type objects, the point is constrained within the rectangular bounds of the text. In the case of grouped objects, the point produced by the **Constrain...** function is constrained to fall within the rectangular bounds of the group. (Note that this can mean that the resulting point falls outside all of the components of the group.)

Among the **Constrain...** functions that constrain to edges of objects (**ConstrainToEdge** and **ConstrainMouseToEdge**), the constraints on simple line and fill objects constrain the point result to the pen edge. In the case of *text* type objects and in the case of grouped objects, the point will be constrained to the edge of the bounding rectangle of the text object or grouped object.

Although the **Constrain...** functions are often used to constrain the **Location** of one object to be within or on the edge of another object, they can be used to produce point values for other purposes as well. The **Constrain...** functions know

nothing about the size of the object whose location is being constrained, so the constrained object is not forced to be entirely within a constraining object when its **Location** is determined by a **ConstrainToFill** function; only its **Location** is constrained to be within the constraining object.

DisplaySize

DisplaySize ()

The function **DisplaySize** returns a point whose **x** and **y** values are the width and height of the display in pixels.

MakePoint

MakePoint (<numberX>, <numberY>)

The function **MakePoint** takes two number parameters. The first is treated as the value of the [X] component of the point result of the function; the second number value is made the value of the [Y] component of the point. This manual contains examples of a number of rules for controlling the motion of objects by using **MakePoint** to determine the value of the objects' **Location** attributes.

MousePosition

MousePosition ()

MousePosition returns the point in the local coordinate system of the object whose attribute has the constraint rule, whenever and as long as the left mouse button is pressed. An author can use **MousePosition** to make an object move with the mouse when it is pressed.

MousePosition

MousePosition (<object>)

When **MousePosition** is called with an object argument, then it returns the point in the coordinate system of the specified parameter object (when the left mouse button is pressed).

PercentToPoint

PercentToPoint (<object>, <number>)

PercentToPoint produces a point that is at a particular proportion of the edge extent of the <object> parameter, as specified by the <number> parameter. <number> should be between 0 and 1. If the object is a line, then the number will determine what proportion of the distance between the starting point of the line and its ending point of the line. This is also true of splines and multilines. If the <number> parameter is 0.25, then the point returned by **PercentToPoint** will be one-fourth of the edge extent of the object.

If the object is a closed figure, such as a rectangle or an ellipse, then the **PercentToPoint** function will generate a point that depends on the type of the object that is the first parameter of the function. A <number> of 0.5 will result in a point on the opposite side of the object. Text objects and group objects constrain the

returned point to the rectangular bounds of the object, just as though there were a rectangle around that object.

Transform

Transform (<from_object>, <point>, <to_object>)

The **Transform** function is used to translate point values from the coordinate system of the *parent* of <from_object>, to the corresponding value as expressed in the coordinate system of the *parent* of <to_object>. The parameters from_object and to_object are expressed as objects, e.g. .sys.Button., instead of as strings, e.g. ".sys.Button." .

For example, consider an object 'A', which is in one group, and an object 'B', of a different group. As always, A's and B's locations are expressed in the coordinate system of their respective parents. Now suppose one wants to write a rule to place object B at A's location. The rule for B's location

```
Transform(<...>.A.,  <...>.A.Location, self)
```

converts A's location (in A's parent's reference system), to the value within B's parent's reference system, such that B would move to precisely where A is in the scene.

TransformStr

TransformStr (<from_object>, <point>, <to_object>)

The **TransformStr** function is used to translate point values from the coordinate system of the *parent* of <from_object>, to the corresponding value as expressed in the coordinate system of the *parent* of <to_object>. The parameters from_object and to_object are expressed as strings, e.g. ".sys.Button.", instead of as objects, e.g. .sys.Button. .

For example, consider an object 'A', which is in one group, and an object 'B', of a different group. As always, A's and B's locations are expressed in the coordinate system of their respective parents. Now suppose one wants to write a rule to place object B at A's location. The rule for B's location

```
TransformStr("<...>.A.",  <...>.A.Location, "<..>.B")
```

converts A's location (in A's parent's reference system), to the value within B's parent's reference system, such that B would move to precisely where A is in the scene.

Pattern Functions

The following function returns values of type pattern.

MakePattern

MakePattern (<text>)

MakePattern is used to create a pattern value, given a descriptive text string. There are a number of different kinds of strings that can be used to make patterns. The simplest way to specify a pattern is to use a text parameter that consists of a number between 0 and 1 enclosed in quotes. This specifies the proportion of fill color that the pattern should have. For example,

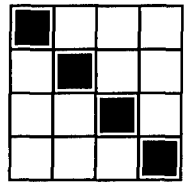
`MakePattern ("0.75")`
will produce a fill that is 75% fill color and 25% pen color.
`MakePattern ("0.33")`
produces a fill that is 33% fill color and 67% pen color. In a sense, this type of pattern is not a pattern at all, but rather a specification for assigning a color that results from combining the official FillColor and LineColor attributes in the proportion specified.

The usage
`MakePattern ("none")`
has the special behavior of filling the object with the pattern "None". This is the pattern that displays underlying objects in the fill area and that does not respond to a mouse click. This pattern is used when an object is to consist of only edge elements. (In reality, any text parameter that begins with the letter "n" will be treated as "none".)

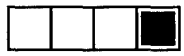
There are three more complex, but sometimes quite useful, ways to specify patterns by encoding the pattern in a text string that consists of hexadecimal numbers. If an author has programming experience, her or she will probably feel comfortable about building patterns using this technique. If the concept of numbers expressed in base 16 is new to an author, assistance will probably be required to build these kinds of pattern descriptions.

The first approach to specifying patterns pixel-by-pixel is to prescribe patterns as matrices of repeating 4-by-4-bit cells. Authors use **`MakePattern`** to describe one of these cells. Here is one such description
`MakePattern ("8421")`
which generates a diagonal lines pattern:

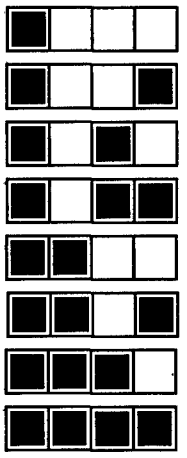
Examined close up, a single 4-by-4-pixel cell of this pattern looks like the grid shown at the right.



This pattern can be viewed as four rows, each with four pixels. In iRides, each such 4-pixel row has a one-character 'name'. There are 16 such patterns. Their names are shown below.

'name'	4-bit row pattern
0	
1	
2	
3	
4	
5	
6	
7	

8
9
a
b
c
d
e
f



Here, the four 'names' for bit patterns "1", "2", "4", "8" are associated with the four rows of the pattern generated by the call **`MakePattern("8421")`**.

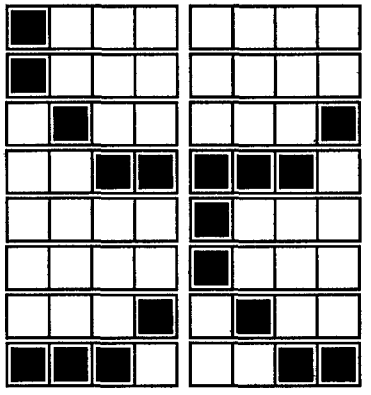
What pattern will be generated by **`MakePattern("edb7")`**?

It is also possible to generate repeating 8-by-8-bit patterns using **`MakePattern`**. One uses the same names that are used for the 4-by-4-bit cells, but enters the names in pairs, where each pair designates a row of eight bits. For example, the call

```
MakePattern ("01 01 82 7c 10 10 28 c7")
```

will generate the kind of fill pattern shown at the right. The larger pattern 8-by-8 cell size makes it possible to create more interesting patterns than those that can be achieved with 4-by-4 cells.

Examining this 'fish scale' pattern up close, reveals that it can be analyzed as filling the grid cells as shown at the right here. The call to **`MakePattern`** has eight pairs of 'names' for 4-bit cell patterns. Each *pair* of names corresponds to a row in the 8-by-8-bit pattern. **Within a row, the 4-bit pattern on the right is named first in the pair.** Thus the first row of the figure shown here is designated by "**`01`**" in the call to **`MakePattern`** shown above.



Authors can even specify patterns as large as 16-by-16 bits using **`MakePattern`**. This is done by using a text argument that consists of sixteen row descriptions. Each row description is a set of four 'names' (from the set "0"... "f" shown above).

```
MakePattern ("0000 59a6 aaaa 5aaa 8aa6 aaaa 59b2 0000 0000
              2cd3 5555 2d55 4553 5555 2cd9 0000")
```

Within each set of four 'names', the first name determines the last four bits of the row, and the last name determines the first four bits of the row.

Universal Operators

The following language constructs, **if...then...else** and **with** can return values of any of the iRides value types: number, logical, text, point, array, object reference, attribute reference. In a sense, there are six **if...then...else** constructs, each of which returns a different value type, and six **with** constructs.

if...then...else...

if <logical>
then <an expression of any type>
else <an expression of the same type>

The **if...then...else...** construct returns the value of the expression following the **then** when the **if** expression evaluates as **true**. In other cases the construct returns the value of the expression that follows the **else**. The **else** portion of this construct is optional; if it is not specified and the **if** expression evaluates to false, nothing further happens.

A less commonly used version of the **if...then...else** construct takes a number expression in the **if** clause rather than a logical expression.

if <number>
then <an expression of any type>
else <an expression of the same type>

This construct has exactly the same meaning as

```
if theNumber = 0
        then <an expression of any type>
        else <an expression of the same type>
```

In other words, the if-expression returns **true** only if it evaluates to zero.

with

with <object>
<an expression of any type>

The **with** construct makes it possible to write more readable rules. Every reference to an attribute or to an object within the scope of the **with** (that is, every reference in the expression that follows the object reference) will be interpreted as being preceded by the object reference.

In the body of the event shown immediately below there is a use of the **with** construct in the third line ("...**with.P.KeyPad.Screen.**"). Every reference in the scope of this **with** statement that does not begin with "." (a period) is to be interpreted as being prefaced with "**.P.KeyPad.Screen.**". For example the **Mode** attribute that is assigned the value "dialing" is an attribute of **.P.KeyPad.Screen**, as are all the **LastDialed** attributes in the most indented block of code below.

```
if (.P.KeyPad.Screen.Mode = "standby" or .P.KeyPad.Screen.Mode =
"recall") and .P.KeyPad.Screen.MidTxt.TextValue <> ""
   then with .P.KeyPad.Screen  {
      Mode := "dialing";
      .P.KeyPad.YesBtn.StartTime := "start call";
      if MidTxt.TextValue <> LastDialed1
         then {
            LastDialed5 := LastDialed4;
            LastDialed4 := LastDialed3;
            LastDialed3 := LastDialed2;
            LastDialed2 := LastDialed1;
            LastDialed1 := MidTxt.TextValue;
            }
      }
```

One consequence of using **with** is that the author must use full path references for the object that owns the behavior-governing rule. The **StartTime** attribute referenced on the fourth line line belongs to **.P.Keypad.YesBtn.**, the owner of this event. It would not ordinarily require a full path specification in an event body, but it requires one because it is in the scope of a **with** that specifies a different object.

case

```
case <case selector>
   {
      <case label> : <an expression of any type>
      <case label> : <an expression of the same type>
      ...
      default : <an expression of the same type>
   }
```

The **case** syntax can sometimes be used in place of a series of nested **if...then...else**-type statements where each condition checks for a different value of the same expression. A **case** statement may be more compact and easier-to-read. Here is a constraint rule for **ValveName**, an attribute of the text type:

```
case ValveNumber
 {
    0 : .Scn.Valve0.DisplayName
    1 : .Scn.Valve1.DisplayName
    2 : .Scn.Valve2.DisplayName
    default : "No valve name available"
 }
```

A **case** construct is like an **if...then...else** that permits more than two possible consequences. As with an **if...then...else** statement, only one of the consequences actually occurs. **case** is a reserved word in iRides. In a **case** statement, the word "case" is immediately followed by the *case selector*, which is **ValveNumber** in the example above. A case selector is an expression of the type number or the type logical.

The case selector can be simply an attribute reference, as in the above example, where **ValveNumber** is a number attribute. A series of cases then follows in the **case** construct. Each case begins with a *case label* (a constant—not an expression) followed by a space and the symbol "**:**" and then a value expression. In the constraint **case** construct, the last case label must be the reserved word **default**, which signifies the case that is to apply if the case selector is not equal to any of the other (non-**default**) case labels. A constraint **case** construct must have a **default** case label because every expression must return a value in iRides.

In the example above, the **case** construct returns a value of the text type, but **case** can be used to return a value of any type. No matter what the return type of the **case** construct, the case selector and the case labels can be any of the five case selector types. Therefore, case selectors can be of type text, as in the example below.

```
case ValveType
{
   "cross": .Scn.Valve0.DisplayName
   "sequence": .Scn.Valve1.DisplayName
   "safety": .Scn.Valve2.DisplayName
   default : "No valve name available"
}
```

If a case selector is of type color, then the case labels will be color constants, such as **[0.763, 0.522, 0.989]**. If the case selector is of type point, then the case labels will be point constants, such as **[220, 382]**. If the case selector is of type logical, there can be only two case labels, **true** and **false**. It doesn't really make sense to use a logical case selector. Instead, an author should use an **if...then...else** construct.

Note: At least one blank space must be inserted between the "**t**" in **default** and the following colon, "**:**".

Event Language Elements

An iRides *statement* is a language construct that carries out an action, but does not return a value. One type of statement is the assignment statement, which uses the assignment operator := to assign values to attributes. Another type of statement is one that uses an event statement function, such as **RingBell()** to carry out an action that cannot be achieved through attribute value manipulations. Statements may appear only in event bodies.

:=

<any attribute> **:=** <any constraint expression of the same type as the attribute>**;**

The **:=** event operator is the *assignment* operator. It can be used to assign to any attribute the value that can be computed from the constraint expression on the right side of the operator. It can also be used to assign a value to an array element, as in

```
Location[Y] := 450;
```

//

// <any text on one line >

// is the standard comment operator. If a line begins with **//**, then iRides treats the remainder of the line as a comment. Each line of comment must be preceded by two slash symbols: **//**.

Each such comment will be given a line to itself in the formatted event body. If an author attempts to append a comment to the end of a statement line, that comment will move to the line below when the event is accepted. A comment in iRides *is* a statement; hence, it can appear only where a statement can appear.

/* */

/* <any text > */

/* ... */ is the standard multiline comment operator. All text between the opening '/*' and the closing '*/' will be treated as a comment, and will be displayed exactly as typed in.

//%

//% <text on one line >

One way to create iRides simulations is to develop them using the latest versions of the RIDES/VIVIDS program, an application written in C++ for Unix-like operating systems. (Rivets, the most recent port of this program, is available for PCs running Linux, for Macintosh computers running OS 10.3 with X11, and for Silicon Graphics running IRIX.) Certain functions in RIDES/VIVIDS are now considered obsolete and are not available in iRides (unless someone would like to recreate them using the features described in Chapter 4 to implement them for inclusion in the sim/user/ directory). Other functions, described in a later section, have been preserved in iRides to support the re-use of previously authored simulations, but are considered deprecated, because there are now better ways to achieve the desired effects. It is possible to use a special approach to commenting to embed iRides-specific code in a Rivets function so that it will be ignored by Rivets, but can still be used in iRides.

//% allows an author to include in a VIVIDS simulation a line of code that is invalid for VIVIDS but valid for iRides. For example, the procedure 'watchmouse' is unknown in VIVIDS; if a VIVIDS event includes a call of this procedure, the event will not be accepted. However, the author may be authoring in VIVIDS but intending to run the event in iRides. To make this possible, this special comment marker is defined. If a VIVIDS event includes the line

```
//% watchmouse(true);
```

VIVIDS will just treat it as a comment line, but when that event is run in iRides, it will be treated not as a comment, but as a call of watchmouse.

It may sometimes be the case that when running an event in VIVIDS, one procedure should be called, but when running that event in iRides, a different procedure should be called. This can be accomplished by combining the '//%' comment marker and the '/* ... */' comment marker.

```
//% playsound("bad.wav"); /*
RingBell();
//% */
```

When running VIVIDS, the first and third lines will be treated as comments, but not the second, so **RingBell** will be called. When running iRides, the first line will not be treated as a comment, so **playsound** will be called, but the '/*' at the end of that line will be treated as the start of a multiline comment, terminated in line three, so **RingBell** will not be called.

case

case <case selector>
{
<case label> **:** <any event statement> **;**
<case label> **:** <any event statement> **;**
...
default : <any event statement>**;**
}

The event **case** construct does not return a value, as do the case constraints, but rather controls the execution of event statements, as in the example below.

```
case .Scn.UserDisplay.ValveNumber
  {
     0: .Scn.UserDisplay.ValveName := .Scn.Valve0.DisplayName;
     1:
        {
           .Scn.UserDisplay.ValveName := .Scn.Valve1.DisplayName;
           RingBell();
        }
     2:
        {
           .Scn.UserDisplay.ValveName := .Scn.Valve2.DisplayName;
           RingBell();
           MoveToBack (.Scn.Valve2.);
        }
     default : RingBell();
  }
```

A series of statements (enclosed in **{ }** characters) can be the <any event statement> that results from a case matching in the **case** event construct. Every statement must end with a **;** character. The **default** case is *not allowed* in the **case** event statement. (However, the **case** constraints *require* that the **default** case be specified.)

Note: At least one blank space must be inserted between the "**t**" in **default** and the following colon, "**:**".

AddToArray

AddToArray (<attributename>,<value>)

This event statement makes it possible to add a value to an array. The first parameter provides the name of the attribute that contains the array, and the second parameter is the value that is to be added. If the attribute does not exist or is not of the type *array*, then a simulation error will occur. Calling this procedure will result in the array's size being increased by one, and the value will be put in the last cell of the array. The value can be of any type—text, numeric, logical, etc.—but it must be of the same type as the values that are already in the array.

DoEvent

DoEvent (<eventname>)

This event statement makes it possible for one event to call another explicitly. Callable events provide a mechanism for developing and executing subroutines, The called event need not have a specified trigger or delay. When a statement with this form is executed, the specified event's body is executed immediately. Even if the called event has a defined delay, that delay will be ignored. The condition part of a called event is also ignored.

Upon completion of the called event body, control returns to the calling event. Called events can be embedded. For example, the body of event A can call event B, which in turn calls event C. When C is finished, control returns to the next statement in event B (after the **DoEvent** call of event C). When B is finished, event A's execution resumes with the statement after its call of event B.

Recursion is **not** allowed in callable events. If an event that is not yet completed is called again, that subsequent call (that **DoEvent** statement) is simply skipped.

DragAndClick

DragAndClick (<objectname>,<objectname>);
DragAndClick (<objectname>,<point>,<scenename>);
DragAndClick (<point>,<objectname>,<scenename>);
DragAndClick (<point>,<point>,<scenename>);

This event statement makes it possible to simulate placing the mouse in one location, clicking the mouse, holding it down while it is moved to a second location, and releasing it. (Unfortunately, the movement of the mouse is not visible.) One use for this would be to simulate moving a slider from one position to another. The first position for placing the mouse is determined by the first parameter, possibly in conjunction with the third. The final position for the mouse is determined by the second parameter, possibly in conjunction with the third. If the first or second parameter is an object name, the point used is the location of that object; if the first or second is a point, then the scene name must be specified, and the mouse position is taken as the value of the point on that scene.

```
DragAndClick(.Front_Panel.Power_Light.,.Front_Panel.Mode_Lig
ht);
```

will move the mouse to the location of the Power_Light, click the mouse, hold it down while moving to the location of the Mode_Light, and release the mouse.

```
DragAndClick([50,50],[100,100],Front_Panel);
```

will move the mouse to the location [50,50] in the Front_Panel scene, click the mouse, hold it down while moving to the location [100,100] in the **Front_Panel** scene, and release the mouse.

FadeIn

FadeIn (<object_name>,<number>);

This event statement makes it possible to make a simulation object gradually appear. The object to be affected is specified with the **object_name** parameter. The length of time, in seconds, that it takes to go from invisible to fully visible is specified with the **number** parameter.

FadeOut

FadeOut (<object_name>,<number>);

This event statement makes it possible to make a simulation object gradually disappear. The object to be affected is specified with the **object_name** parameter. The length of time, in seconds, that it takes to go from fully visible to invisible is specified with the **number** parameter.

fclose

fclose (<filename>);

The file to close is specified by the text value of <filename>. This function can be used only as an event statement.

Flare

Flare (<attributename>,<value>,<delay>);

This event statement makes it possible to wait for a specified amount of time before setting an attribute value. For example, it might be useful to specify that a particular light should come on 10 seconds after some other event. This could be done in an event by writing

```
Flare(".Front_Panel.warningLight.state","on",10);
```

The first parameter is the name of the attribute whose value is to be set. The second parameter is the value to be used; it may be of any type—logical, numeric, etc. The third parameter is the length of the delay, in seconds, before the value change is to take place.

fopen

fopen (<filename>, <type>);

The file to open is specified by the text value of <filename>. The <type> argument can be any text expression that evaluates to either "**r**", "**w**", "**a**" or "**r+**". Type "**r**" opens a file for reading. "**w**" opens a file for writing. "**a**" opens a file for writing, but all output is appended to the existing file if there is one. "**r+**" opens a file for random access. This function can be used as either an event statement, or as an expression

returning a boolean value. A return value of **true** means the file opening was successful.

If the file will be written to using the **SetNthRecord** function, then **fopen** should be opened with the special read/write parameter "**r+**", as in

```
fopen ("MyRecords", "r+");
```

if...then

```
if <logical>
   then <any event statement>;
```

An **if...then** *statement* does not require an **else** clause. The substatement in the **then** clause must end in a semicolon.

if...then...else

```
if <logical>
   then <any event statement>;
   else <any event statement>;
```

Although it is superficially quite similar to the **if...then...else** expression construct, the iRides **if...then...else** statement construct does not return a value. Instead, when the event is triggered, if the value of the **if** expression is true then the event statement following the **then** is carried out. Otherwise the event carries out the event statement following the **else**.

The **if...then...else** statement requires a semicolon after each statement. Also, compound statements are permitted, as in this example:

```
if .S.Pwr.VoltsOut <= .S.Pwr.MaxSafeVolts
   then {
        .S.Pwr.VoltsAdjustUp := .S.Pwr.VoltsAdjustUp + 1;
        .S.Pwr.OldVolts := .S.Pwr.VoltsOut;
      }
   else RingBell();.
```

MoveAndClick

```
MoveAndClick (<objectname>);
MoveAndClick (<point>,<scenename>);
```

This event statement makes it possible to simulate moving the mouse to a specified location and clicking it. (Unfortunately, the movement of the mouse is not visible.) One possible use for this would be to simulate clicking a button. The ending position for the mouse is determined by the first parameter, possibly in conjunction with the second. If the first parameter is an object name, the point used is the location of that object. If the first parameter is a point, then the scene name must be specified, and the mouse position is taken as the value of the point on that scene.

```
MoveAndClick(.Front_Panel.Power_Toggle.);
```

will move the mouse to the location of the Power_Toggle and click the mouse.

```
MoveAndClick([50,50],Front_Panel);
```

will move the mouse to the location [50,50] in the **Front_Panel** scene and click the mouse.

MoveBwd

MoveBwd (<object>);

The **MoveBwd** event statement moves the object designated by the parameter one step closer to the back in the list of objects in its parent's view.

MoveFwd

MoveFwd (<object>);

This event statement has the effect of moving the object designated by the parameter one step closer to the front in the list of objects on the scene.

MoveToBack

MoveToBack (<object>);

The **MoveToBack** event statement moves the object argument to be the backmost object within its parent's view. The object will then "lie below" other objects that it may have formerly obscured.

MoveToFront

MoveToFront (<object>);

The **MoveToFront** event statement moves the object argument to be frontmost within its parent's view. The object will then "lie on top of" other objects that may have formerly obscured it.

Play_MPEG

Play_MPEG (<text>);
Play_MPEG (<text>,<logical>);

Play_MPEG (**"TheFileName"**); opens a video playing interface with the mpeg file specified in the text parameter to **Play_MPEG**. There is an optional second parameter. If it is set to **true**, the simulation is inactive until the video finishes; if it is **false**, the simulation remains active. The default value is **true**. The user can use the graphic widgets of this interface to start and stop video playing of the named file.

PlayInstruction

PlayInstruction (<text>, <text>, <text>);

PlayInstruction is used to present instruction under the control of an event. Authors can write events that deliver all or only a portion of a previously authored lesson to students when a particular condition occurs. The event condition that triggers the presentation of instruction may refer to some state of the simulation, or it may refer to student model attribute value changes or instructional item attributes.

The **PlayInstruction** function can only be used in event bodies. The function takes three parameters. The first text parameter is the name of the instructional item to play. An example is

"I:HandlePowerError.We_will_reset."

The second text parameter is one of "Demonstrate", "Practice", or "Test". This parameter prescribes the mode in which the lesson or lesson fragment should be presented. The third parameter is either "Resume" or "Stop". The "Resume" option means that if opportunistic instruction occurs during a lesson, then that lesson should resume when the opportunistic lesson or lesson fragment is finished. The "Stop" option means that the lesson that had been playing will end right away.

PlaySound

PlaySound (<text>);

PlaySound (**"TheFileName"**); takes a text parameter, which specifies the name of the file in .au format that should be played. If a sound is already being played when another **PlaySound** command is issued, then the old sound will stop playing and the new one will begin.

PopupCheckBox

PopupCheckBox
(<attribute_name>,<prompt>,<numberX>,<numberY>,<scene_name>,
<button_label>,<logical>,...);

PopupCheckBox opens a Java Swing interface with two or more check boxes. It takes a minimum of seven parameters. The interface includes an "OK" button. When that button is selected, the labels of the check boxes that were selected are returned to the simulation. The first parameter specifies the name of the attribute that is to receive that returned value. The second parameter, <prompt>, contains text that will be displayed in the interface above the check boxes; its value could be "". The third, fourth, and fifth parameters specify where the interface is to be displayed, at position [<numberX>,<numberY>] in scene <scene_name>. If <scene_name> is specified as "", the location is understood to be relative to the simulation frame, not to any specific scene. The remaining parameters should be as many text-logical pairs as the author needs, with a minimum of one pair. The first element of the pair is the label that should be placed on a check box and the second element specifies whether that checkbox should be initialized as selected (true) or not selected (false). An example of the use of this procedure is

```
PopupCheckBox(".sys.valuesToShow","Which values should be made visible?",0,0,"","Probability Values",true,"Utility Values",true,"Utility Sums",false);
```

PopupDialog

PopupDialog
(<attribute_name>,<prompt>,<numberX>,<numberY>,<scene_name>);

PopupDialog opens a Java Swing dialog, for presentation purposes only. It takes five parameters. The interface includes an "OK" button. When that button is selected, the dialog is dismissed and the value "CLOSED" is returned to simulation. The first parameter specifies the name of the attribute that is to receive that returned value. The second parameter, <prompt>, contains text that will be displayed in the interface. The third, fourth, and fifth parameters specify where the dialog is to be displayed, at position [<numberX>,<numberY>] in scene <scene_name>. If <scene_name> is specified as "", the location is understood to be relative to the simulation frame, not to any specific scene.

PopupKeypad

PopupKeypad
(<attribute_name>,<prompt>,<numberX>,<numberY>,<scene_name>,
<initial_value>**);**

PopupKeypad opens a Java Swing keypad interface. It takes six parameters. The interface includes an "OK" button. When that button is selected, the value that was entered into the keypad is returned to the simulation. The first parameter specifies the name of the attribute that is to receive that returned value. The second parameter, <prompt>, contains text that will be displayed at the top of the interface; its value could be "". The third, fourth, and fifth parameters specify where the interface is to be displayed, at position [<numberX>,<numberY>] in scene <scene_name>. If <scene_name> is specified as "", the location is understood to be relative to the simulation frame, not to any specific scene. The sixth parameter is a number that specifies an initial value for the keypad. An example of the use of this procedure is

```
PopupKeypad(".sys.keypadVal","What is the area of a 2' x 3' rectangle?",0,0,"",0);
```

PopupLogin

PopupLogin
(<attribute_name>,<prompt>,<numberX>,<numberY>,<scene_name>,
<initial_value>,<initial_value>**);**

PopupLogin opens a Java Swing login interface. It takes seven parameters. The interface includes an "OK" button. When that button is selected, the values that were entered into the interface are returned to the simulation. The first parameter specifies the name of the attribute that is to receive those returned values. The second parameter, <prompt>, contains text that will be displayed at the top of the interface; its value could be "". The third, fourth, and fifth parameters specify where the interface is to be displayed, at position [<numberX>,<numberY>] in scene <scene_name>. If <scene_name> is specified as "", the location is understood to be relative to the simulation frame, not to any specific scene. The sixth parameter specifies an initial name for the login interface. The seventh parameter specifies an initial password. The password will be displayed with a series of '*',s. An example of the use of this procedure is

```
PopupLogin (".sys.textVal","Please log in your user name and password.",0,0,"","Joe Smith","1234");
```

PopupMenuEntry

PopupMenuEntry
(<attribute_name>,<prompt>,<numberX>,<numberY>,<scene_name>,
<menu_item>,...**);**

PopupMenuEntry opens a Java Swing interface with one or more menu items. It takes a minimum of six parameters. When a menu item is selected, that item is returned to the simulation. The first parameter specifies the name of the attribute that is to receive that returned value. The second parameter, <prompt>, contains text that will be displayed in the interface above the menu; its value could be "". The

third, fourth, and fifth parameters specify where the interface is to be displayed, at position [<numberX>,<numberY>] in scene <scene_name>. If <scene_name> is specified as "", the location is understood to be relative to the simulation frame, not to any specific scene. The remaining parameters should be as many text items as the author needs for the menu, with a minimum of one. An example of the use of this procedure is

```
PopupMenuEntry(".sys.valuesToShow","Which color is not in the French flag?",0,0,"","red","green","blue");
```

PopupNumericEntry

PopupNumericEntry
(<attribute_name>,<prompt>,<numberX>,<numberY>,<scene_name>,
<logical>,<lower_limit>,<upper_limit>**);**

PopupNumericEntry opens a Java Swing text entry interface. It takes eight parameters. The interface includes an "OK" button. When that button is selected, the value that was entered into the interface is returned to the simulation. The first parameter specifies the name of the attribute that is to receive that returned value. The second parameter, <prompt>, contains text that will be displayed at the top of the interface; its value could be "". The third, fourth, and fifth parameters specify where the interface is to be displayed, at position [<numberX>,<numberY>] in scene <scene_name>. If <scene_name> is specified as "", the location is understood to be relative to the simulation frame, not to any specific scene. The sixth parameter specifies whether a floating point entry is to be allowed (true) or not (false). The seventh and eighth parameters set the lower and upper limits on the values that will be accepted. If the user types in anything that does not conform to these specifications, the type in is ignored. An example of the use of this procedure is

```
PopupNumericEntry (".sys.textVal","Enter a number between -5 and +5, inclusive.",0,0,"",false,-5,5);
```

PopupRadio

PopupRadio
(<attribute_name>,<prompt>,<numberX>,<numberY>,<scene_name>,
<initial_value>,<button_label>,<button_label>,...**);**

PopupRadio opens a Java Swing interface with two or more radio buttons. It takes a minimum of eight parameters. The interface includes an "OK" button. When that button is selected, the label of the radio box that was selected is returned to the simulation. The first parameter specifies the name of the attribute that is to receive that returned value. The second parameter, <prompt>, contains text that will be displayed in the interface above the radio buttons; its value could be "". The third, fourth, and fifth parameters specify where the interface is to be displayed, at position [<numberX>,<numberY>] in scene <scene_name>. If <scene_name> is specified as "", the location is understood to be relative to the simulation frame, not to any specific scene. The sixth parameter is a string that indicates which radio button is to be initially selected; if it doesn't match any of the button labels, nothing will be preselected. The remaining parameters should be as many text items as the author needs, with a minimum of two. Each of these items is a label that should be placed on a radio button. An example of the use of this procedure is

```
PopupRadio(".sys.valuesToShow","Who was the third president of the United States?",0,0,"","Thomas Jefferson","George
```

```
Washington,"Thomas Jefferson","Abraham Lincoln","Franklin
Delano Roosevelt");
```

PopupSlider

PopupSlider
(<attribute_name>,<prompt>,<numberX>,<numberY>,<scene_name>,
<lower_limit>,<upper_limit>,<initial_value>);

PopupSlider opens a Java Swing text entry interface. It takes eight parameters. The interface includes an "OK" button. When that button is selected, the value of the slider is returned to the simulation. The first parameter specifies the name of the attribute that is to receive that returned value. The second parameter, <prompt>, contains text that will be displayed at the top of the interface; its value could be "". The third, fourth, and fifth parameters specify where the interface is to be displayed, at position [<numberX>,<numberY>] in scene <scene_name>. If <scene_name> is specified as "", the location is understood to be relative to the simulation frame, not to any specific scene. The sixth and seventh parameters set the lower and upper limits on the slider. The eight parameter specifies an initial value for the slider. These last three values should be integers.

```
PopupSlider (".sys.textVal","Select an initival value for
the voltage.",0,0,"",0,28,0);
```

PopupTextEntry

PopupTextEntry
(<attribute_name>,<prompt>,<numberX>,<numberY>,<scene_name>,
<initial_value>);

PopupTextEntry opens a Java Swing text entry interface. It takes six parameters. The interface includes an "OK" button. When that button is selected, the value that was entered into the interface is returned to the simulation. The first parameter specifies the name of the attribute that is to receive that returned value. The second parameter, <prompt>, contains text that will be displayed at the top of the interface; its value could be "". The third, fourth, and fifth parameters specify where the interface is to be displayed, at position [<numberX>,<numberY>] in scene <scene_name>. If <scene_name> is specified as "", the location is understood to be relative to the simulation frame, not to any specific scene. The sixth parameter specifies an initial value for the interface. An example of the use of this procedure is

```
PopupTextEntry (".sys.textVal","What is your
name?",0,0,"","");
```

PressClose

PressClose();

This event function has the effect of clicking on the *Close* button in the student's instruction window. The *Close* button is used to close the student window. It re-opens automatically the next time text is presented in that window.

PressContinue

PressContinue();

The **PressContinue** function has the effect of a student clicking on the *Continue* button in the instruction window. The *Continue* button makes certain instructional items continue. A *Wait for Student* instructional item, for example, is completed when the student clicks the *Continue* button. A *Goal* instructional item is evaluated by iRides when the student chooses this button.

PressDontKnow

PressDontKnow();

This event function has the same effect as a Student pressing the *Don't Know* button in the instruction window. The *Don't Know* button makes iRides present the item remediation for the current instructional item.

PressStop

PressStop();

This function has the same effect, when called, as a student pressing the *Stop* button in the instruction window interface. The *Stop* button ends the current lesson but does not end the course presentation.

Print

Print (<any expression>);

The **Print** event statement will produce output to the term window. The expression is optional; if used, it can be any expression that returns a value.

Print(); will just output a blank line to the terminal.

Print(4 * 3); will output "12".

Print(format(.....)); will output whatever the format call produces.

Quit

Quit (<logical>);

The **Quit** event statement will cause the iRides application to quit. If the argument is **true**, a dialog box will pop up to confirm the quit. If the user chooses "No", iRides continues. A **false** argument will simply quit without asking for confirmation. This makes it possible for authors to decide whether quitting must be confirmed for their application.

RandSeed

RandSeed (<number>);

The **RandSeed** event statement function initializes the random number generator. Using the same number with **RandSeed** ensures that the same sequence of random numbers will appear on different runs in a simulation. To get different sequences, one must initialize with a different number each time. For example, one could use this call in an event body:

RandSeed(Now());

Refresh

Refresh ();

The event statement **Refresh()** will cause all graphics to be updated on the screen. iRides automatically updates the graphics once an event is finished; this statement allows the update to occur at specified points *within* an event. Authors need to remember that there is no way to accept new user input during an event, and that the clock is frozen during the event.

RemoveFromArray

RemoveFromArray (<attributename>,<value>)

This event statement makes it possible to remove a value from an array. The first parameter provides the name of the attribute that contains the array, and the second parameter is the value that is to be removed. If the attribute does not exist, a simulation error will occur. If the value is not found in the array, the call is ignored.

ResetClock

ResetClock ();

The **ResetClock** event statement function resets the internal clock. The value of **.sys.Clock** immediately changes to 0 and begins incrementing again when the event is over. If an event contains a number of statements, all the statements that follow the **ResetClock** statement in the event will take place before the clock takes on a value greater than 0.

ResetClock

ResetClock (<number>);

The **ResetClock** event statement function resets the internal clock. The value of **.sys.Clock** immediately changes to the value of the argument and begins incrementing again when the event is over. If an event contains a number of statements, all the statements that follow the **ResetClock** statement in the event will take place before the clock takes on a value greater than the value of the argument.

RingBell

RingBell ();

Ringbell(); makes the computer beep.

Set_Array

Set_Array (<text>, <array>);

The **Set_Array** event statement sets the value of the attribute specified in the first argument to the value of the second argument. If the attribute does not exist or is not of the type *array*, then a simulation error will occur. Authors should test for the existence of the attribute that is about to be set by using the **AttExists** function.

Set_AttRef

Set_AttRef (<text>, <attref>);

The **Set_AttRef** event statement sets the value of the attribute specified in the first argument to the value of the second argument. If the attribute does not exist or is not of the type *attribute reference*, then a simulation error will occur. Authors should test for the existence of the attribute that is about to be set by using the **AttExists** function.

Set_ObjectRef

Set_ObjectRef (<text>, <objectref>);

The **Set_ObjectRef** event statement sets the value of the attribute specified in the first argument to the value of the second argument. If the attribute does not exist or is not of the type *object reference*, then a simulation error will occur. Authors should test for the existence of the attribute that is about to be set by using the **AttExists** function.

Set_Value

Set_Value (<text>, <value>);

The **Set_Value** event statement sets the value of the attribute specified in the first argument to the value of the second argument. If the attribute does not exist, a simulation error will occur. The value may be of any type—text, number, array, etc. Authors should test for the existence of the attribute that is about to be set by using the **AttExists** function.

SetCursor

SetCursor (<number>);

The **SetCursor** event statement changes the simulation cursor. The cursors used are those from the Xcursor font. The argument must be an even integer between 0 and 152 inclusive, any other value will cause the cursor to change to the default iRides simulation cursor (number 60). To see the appearances of the standard cursors in X, see "Appendix I: The Cursor Font" in *Xlib Reference Manual*, pp 641-642.

SetNthRecord

SetNthRecord (<filename>, <line number>, <text>);

This event statement writes a record to a file. The file must be opened with the read/write argument "r+", as in

```
fopen(<filename>, "r+");
```

When executed, a **SetNthRecord** statement will replace the line of the file specified by <line number> with the text specified by <text>. If <text> is longer than the existing line, then the new record will be truncated to the length of the existing line. If the text length is less than the existing line length, then the line will be padded with new spaces.

If the file contains N lines and the <line number> argument given to **SetNthRecord** is N + 1, then a new line will be added as the N+1 record and will include the whole of <text> (not truncated). The permanent line length of the N+1 record will be established as the length of <text>.

There is also a function version of the **SetNthRecord** routine, which returns a text value that describes the error status resulting from the call. This function returns "** EOF **" if the <line number> parameter has a value greater than the number of lines in the file plus one. It returns "** NOT OPEN **" if the file is not opened with write access. If the function is called without error, it returns the empty string, "".

ShowURL

ShowURL(<text>, <logical>**);**

This event function, when called, will show the URL specified in the first argument. If the second argument is **true**, it will open a new browser window. Otherwise, the page will be shown in an existing browser window.

StartTimer

StartTimer();
StartTimer(<number>**);**

This event statement starts a timer. A possible use for this is to measure how much time passes before something occurs. It should be used in conjunction with StopTimer, which was described earlier in this document. There are 10 different timers available, numbered from 0 to 9. You can specify which timer to use by specifying a value for <number>; if you don't specify one, the default value of 0 will be used. An example of the use of **StartTimer** and **StopTimer** in an event body is

```
Rotation := 0;
StartTimer(3);
While (Rotation < 90)
   Rotation := Rotation + 2;
Print("Rotation completed in ",StopTimer(3)," seconds.");
```

If **StartTimer** is called with some parameter, say '3', and then it is called again with the same parameter, it restarts with a time value of 0. So a subsequent call of **StopTimer** with that parameter will return the time elapsed from the more recent **StartTimer** call.

StopSound

StopSound ();

StopSound (); has the effect of stopping the playing of the current sound.

system_call

system_call (<text>);

This new language feature makes it possible to issue system calls from iRides. The **system_call** function lets an iRides application make any Unix system call. This

makes it possible for sophisticated authors who are also programmers to write their own utility programs that can be called by their iRides simulations.

WatchMouse

WatchMouse (<logical>);

By default, the attribute **.sys.MousePos** only reflects the mouse position when the mouse button is down. If an event calls **WatchMouse**(true), **.sys.MousePos** will reflect the mouse position regardless of the button state. To revert to the default condition, call **WatchMouse**(false);

while

while <logical>
<any event statement>;

This potentially dangerous statement type can be used to repeat a sequence of operations so long as a logical condition holds true. It is very important that the author write the statement in such a way that the logical eventually becomes false as a result of the operations performed in the event statement. If it never does, iRides will never exit the **while**. The only way out in such a case is to kill iRides.

The **while** event statement can be easily misunderstood. Authors must remember that events take place 'instantaneously' within an event. Despite the normal meaning of the word 'while' in ordinary language use, there can be no simulation updating inside the body of a **while** statement. Nothing can 'happen' in a simulation during the execution of a **while**.

writeline

writeline (<filename>, <text>);

This event statement will write out to file the value of the second argument which may be an expression. A newline character will be appended to the text in the output file. There is also a function version of the **writeline** routine, which returns **true** or **false**, depending on whether the functions successfully wrote the text to the file.

Database Functions

In some iRides applications, authors may want to maintain a database of records in which each record can be individually read or written. iRides provides simple mechanisms for retrieving and setting record values in a file on disk. The disk files are text files in which each line is treated as another record. Records are of type text. (Of course, authors can translate portions of textual records into other types, such as numbers, points, and so on.) Records can be of different lengths, but the length of a record cannot be changed. When a record is written with fewer characters than it had before, the remainder of the record is padded out with spaces. When a record is written with more characters than it had before, the extra characters are chopped off and do not appear in the saved record.

fclose

fclose (<filename>);

The file to close is specified by the text value of <filename>. This function can be used only as an event statement.

fopen

fopen (<filename>, <type>);

The file to open is specified by the text value of <filename>. The <type> argument can be any text expression that evaluates to either "**r**", "**w**", "**a**" or "**r+**". Type "**r**" opens a file for reading. "**w**" opens a file for writing. "**a**" opens a file for writing, but all output is appended to the existing file if there is one. "**r+**" opens a file for random access. This function can be used as either an event statement, or as an expression returning a boolean value. A return value of **true** means the file opening was successful.

If the file will be written to using the **SetNthRecord** function, then **fopen** should be opened with the special read/write parameter "**r+**", as in

```
fopen ("MyRecords", "r+");
```

GetNthRecord

GetNthRecord (<filename>, <record number>);

This event statement returns the Nth line from a file. The file must have already been opened with the fopen function. If the **GetNthRecord()** function is called on a file that has not been opened, then the string "** NOT OPEN **" is returned. If the line does not exist in the file, "** EOF **" is returned.

NumRecords

<number> **NumRecords** (<text>);

The function **NumRecords** is a function of type number that returns a count of the records in the file. If the file doesn't exist or isn't open for reading, **NumRecords** will return -1.

readline

readline (<filename>)

This expression will read the next line from the named file if the file is open for reading. If the file is not open for reading, then the function will return the string "*** FILE NOT OPEN ***". The string returned does not include the newline character found in the file. If no more lines are available in the file, then the function returns "*** EOF ***". A well-designed event that reads from a file should check for these values.

SetNthRecord

SetNthRecord (<filename>, <line number>, <text>);

This event statement writes a record to a file. The file must be opened with the read/write argument "r+", as in

```
fopen(<filename>, "r+");
```

When executed, a **SetNthRecord** statement will replace the line of the file specified by <line number> with the text specified by <text>. If <text> is longer than the existing line, then the new record will be truncated to the length of the existing line. If the text length is less than the existing line length, then the line will be padded with new spaces.

If the file contains N lines and the <line number> argument given to SetNthRecord is N + 1, then a new line will be added as the N+1 record and will include the whole of <text> (not truncated). The permanent line length of the N+1 record will be established as the length of <text>.

There is also a function version of the SetNthRecord routine, which returns a text value that describes the error status resulting from the call. This function returns "** EOF **" if the <line number> parameter has a value greater than the number of lines in the file plus one. It returns "** NOT OPEN **" if the file is not opened with write access. If the function is called without error, it returns the empty string, "".

writeline

writeline (<filename>, <text>);

This event statement will write out to file the value of the second argument which may be an expression. A newline character will be appended to the text in the output file. There is also a function version of the **writeline** routine, which returns **true** or **false**, depending on whether the functions successfully wrote the text to the file.

SCORM Functions

iRides simulations can be delivered as Java applications, as Java Web Start applications, and as Java applets. If the applet choice is made, it is possible to make your iRides simulation compliant with the Shareable Content Object Reference

Model (SCORM) of the Advanced Defense Learning (ADL) initiative. To support SCORM-compliant authored simulations, the iRides language has been given four functions that support the required functionality. If any of these functions is called in an execution environment that is not SCORM-enabled, then it will fail silently, as thought it had never been called.

scormfinish

scormfinish ();

This calls the corresponding SCORM function LMSfinish.

scormgetvalue

scormgetvalue (<text>)**;**

This calls the corresponding SCORM function LMSgetvalue, passing it the text parameter.

scorminitialize

scorminitialize ();

This calls the corresponding SCORM function LMSinitialize.

scormsetvalue

scormsetvalue (<text>,<text>)**;**

This calls the corresponding SCORM function LMSsetvalue, passing it the two text parameters

Deprecated Functions and Procedures

Several functions and procedures that were used in VIVIDS are no longer needed in iRides, because iRides offers improved ways to bring about the desired results. Nonetheless, in order to support backward compatibility with VIVIDS, because these language features are still working in iRides, even VIVIDS simulations that relied upon them can be exported and used in iRides. We recommend against using these features in the development of new iRides simulations, however. Many of these functions were required in earlier systems that did not support attribute reference values. Now that one attribute can store a reference to another attribute, there are better (safer, more maintainable) ways to accomplish the desired result.

AttExists

<logical> AttExists (<text>)

This returns a logical value specifying whether or not the attribute named in the <text> parameter is an existing attribute. The attribute name must be fully specified

e.g. **AttExists(".scene1.object.rotation")**. Any event that makes use of any of the ***Lookup_*** or ***Set_*** functions—for example, ***Lookup_Number()*** or ***Set_Logical()***— should first use ***AttExists*** to ensure that there will be no run-time error that could interrupt iRides.

GetKUText

<text> GetKUText (<text>,<text>)

The first argument of the **GetKUText** function specifies the full name of an object with an associated knowledge unit. The second argument specifies the name of a topic for that knowledge unit. The function returns the text of the discussion with that topic name for that object's authored knowledge.
If the object does not exist, or if it has no knowledge unit, or if the knowledge unit does not have the named topic, then the function returns the empty text string, "".
Call this function only within an event.

Lookup_Color

<array of three numbers> Lookup_Color (<text>);

This function returns the color value (an array of three numbers) of the attribute named as the text parameter. If the attribute does not exist, then the simulation will halt giving an "unknown attribute" error. To avoid this, authors should always test for the existence of the attribute first, using the **AttExists** function. For example,

```
if AttExists(format(".S.Light%%" .S.CurNum, ".FillColor")
  then .S.CurColor := Lookup_Color (format(".S.Light%%"
.S.CurNum, ".FillColor"));
```

The effect of this example will be to put into **.S.CurColor** the value that is in some attribute such as **.S.Light21.FillColor**. If the attribute does exist, but is not of the type *color*, then a "wrong type" error will result.
Call this function only within an event.

Lookup_Logical

<logical> Lookup_Logical (<text>);

This function returns the logical value of the attribute named as the text parameter. If the attribute does not exist, then the simulation will halt giving an "unknown attribute" error. To avoid this, authors should always test for the existence of the attribute first, using the **AttExists** function. If the attribute does exist, but is not of the type *logical*, then a "wrong type" error will result. Other such functions for text, points, color, number, and pattern also exist.
Call this function only within an event.

Lookup_Number

<number> Lookup_Number (<text>);

This function returns the numeric value of the attribute named as the text parameter. If the attribute does not exist, then the simulation will halt giving an "unknown attribute" error. To avoid this, authors should always test for the existence of the attribute first, using the **AttExists** function. If the attribute does exist, but is not of the type *number*, then a "wrong type" error will result. Other such functions for text, points, logical, color and pattern also exist.

Call this function only within an event.

Lookup_Pattern

<pattern> Lookup_Pattern (<text>);

This function returns the pattern value of the attribute named as the text parameter. If the attribute does not exist, then the simulation will halt giving an "unknown attribute" error. To avoid this, authors should always test for the existence of the attribute first, using the **AttExists** function. If the attribute does exist, but is not of the type *pattern*, then a "wrong type" error will result. Other such functions for text, points, logical, color and number also exist.
Call this function only within an event.

Lookup_Point

<array of 2 numbers> Lookup_Point(<text>);

This function returns the point value (an array of two numbers) of the attribute named as the text parameter. If the attribute does not exist, then the simulation will halt giving an "unknown attribute" error. To avoid this, authors should always test for the existence of the attribute first, using the **AttExists** function. If the attribute does exist, but is not of the type *point*, then a "wrong type" error will result. Other such functions for text, number, logical, color and pattern also exist.
Call this function only within an event.

Lookup_Text

<text> Lookup_Text (<text>);

This function returns the text value of the attribute named as the text parameter. If the attribute does not exist, then the simulation will halt giving an "unknown attribute" error. To avoid this, authors should always test for the existence of the attribute first, using the **AttExists** function. If the attribute does exist, but is not of the type *text*, then a "wrong type" error will result. Other such functions for number, point, logical, color and pattern also exist.
Call this function only within an event.

QuitVivids

QuitVivids (<logical>);

The **QuitVivids** event statement will cause the iRides application to quit. If the argument is **true**, a dialog box will pop up to confirm the quit. If the user chooses "No", iRides continues. A **false** argument will simply quit without asking for confirmation. This makes it possible for authors to decide whether quitting must be confirmed for their application.

Set_Color

Set_Color (<text>, <color>);

The **Set_Color** event statement sets the value of the attribute specified in the first argument to the value of the second argument. If the attribute does not exist or is not of the type *color*, then a simulation error will occur. Authors should test for the

existence of the attribute that is about to be set by using the **AttExists** function. For example,

```
if AttExists(format(".S.Reflectn%%" .S.CurNum, ".FillColor")
  then Set_Color (format(".S.Reflectn%%" .S.CurNum,
".FillColor")) := .S.CurColor;
```

The effect of this example will be to assign the color value in **.S.CurColor** to some attribute such as **.S.Reflectn21.FillColor**.

Set_Logical

Set_Logical(<text>, <logical>);

This event statement sets the value of the attribute specified in the first argument to the value of the second argument. If the attribute does not exist or is not of the type *logical*, then a simulation error will occur. Authors should test for the existence of the attribute that is about to be set by using the **AttExists** function.

Set_Number

Set_Number (<text>, <number>);

This event statement will set the value of the attribute specified as the first argument to the value of the second argument. Again, if the attribute does not exist or if the attribute is of the wrong type, then a simulation error will occur. Other such functions exist for the other types.

Set_Pattern

Set_Pattern (<text>, <pattern>);

The **Set_Pattern** event statement sets the value of the attribute specified in the first argument to the value of the second argument. If the attribute does not exist or is not of the type *pattern*, then a simulation error will occur. Authors should test for the existence of the attribute that is about to be set by using the **AttExists** function.

Set_Point

Set_Point (<text>, <point>);

The **Set_Point** event statement sets the value of the attribute specified in the first argument to the value of the second argument. If the attribute does not exist or is not of the type *point*, then a simulation error will occur. Authors should test for the existence of the attribute that is about to be set by using the **AttExists** function.

Set_Text

Set_Text (<text>, <text>);

The **Set_Text** event statement sets the value of the attribute specified in the first argument to the value of the second argument. If the attribute does not exist or is not of the type *text*, then a simulation error will occur. Authors should test for the existence of the attribute that is about to be set by using the **AttExists** function.

SetKUText

<logical> SetKUText(<text>,<text>,<text>)

The first argument of the **SetKUText** function specifies the full name of an object with an associated knowledge unit. The second argument specifies the name of a topic for that knowledge unit. The third argument specifies the new topic text. The function returns **true** if the object exists, and **false** if it does not. On success, it will return **true**.

Call this function only within an event.

Older VIVIDS Functions Not Supported in iRides

The following functions were used in VIVIDS but are not supported in iRides.

GetBody
GetDelay
GetTest
SetDelay
SetField
SetName
DefineConfiguration
DeleteConfiguration
InstallConfiguration
NetworkConnect
Remote_Color
Remote_Logical
Remote_Number
Remote_Pattern
Remote_Point
Remote_Text
TScriptLink
TScriptSend
VNetLink
VNetSend

4

Extending the Simulation Language

This chapter is intended for experienced Java programmers who are interested in extending the native function set of the iRides simulation language. It is possible to write new functions that will be utilized automatically by the iRides simulation engine. Programmers create these functions by creating a new Java class that is a subclass of *JRFunction*. Two to four of the methods of JRFunction must be overridden in the new function to make the new function behave appropriately when it is invoked in an author's simulation rules.

The Java runtime environment supports reflection. This makes it possible for a Java class to obtain the names of all its members. Classes (compiled objects) that support reflection must implement a getName method that returns the name of the object. Using reflection, the iRides simulation engine can find and make use of code that was developed independently, perhaps even long after the engine was.

Our approach to making a constraint-based programming language extensible requires that all functions and procedures in the language be derived from a class that implements four special methods: *getFormat*(), *secretTriggers*(), *stayDirty*(), and *compute*(). These behaviors are further explicated below.

When a developer wishes to extend the constraint programming language, he or she must create a new class derived from the constraint language function base class. This derived class must specify the behavior of the new function. It must also specify the types of the parameters of the new constraint language function, by overriding getFormat(). In addition, it may override one or both of the special methods secretTriggers() and stayDirty(). The developer puts the new class file in the directory btl/sim/user/, where it will be automatically found and utilized as required at run time.

Parsing Constraints with 'Native' Function Calls

During the process of parsing the data specifications for a constraint-based interpretable application , the parser encounters function calls that were pre-defined. Because these functions are written in a more efficient programming language than the constraint language itself, which is interpreted, such native functions can run much more quickly than can functions written in the interpreted language. Rather than restricting user- developers to a set of pre-defined native functions, this system permits authors to add their own. At parse time, the system carries out a set of operations each time that it encounters a native function. See Figure 5.

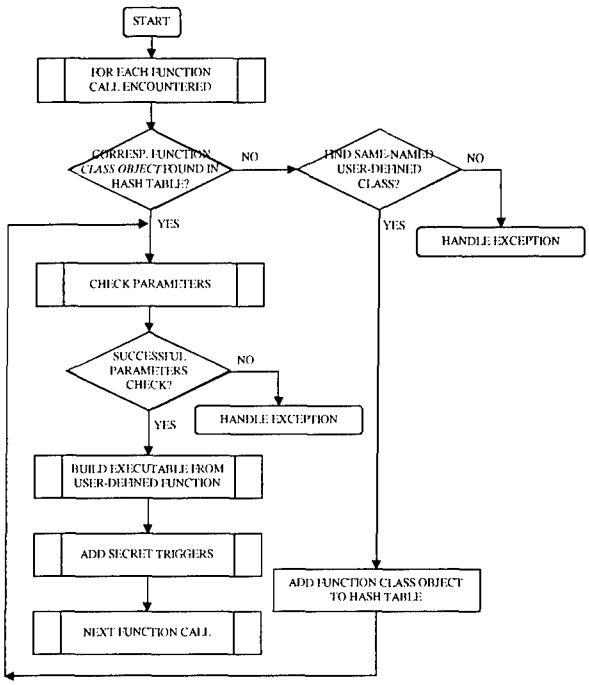


Figure 5. Parsing Constraints with 'Native' Function Calls

First it checks to see whether it already knows about the function, by looking it up in a hash table of all the functions thus far utilized during this session. If the function is not present, then it is found, using reflection, in a set of classes for defined functions. All of these functions are objects that are derived from a base *JRFunction* class. (Similarly procedures are all derived from a base *JRProcStatement* class.) This class implements a number of utility methods and the four methods *getFormat*(), *secretTriggers*(), *stayDirty*(), and *compute*(), which can be overridden by the user- developer.

The parameters of the user-defined function are then checked. This process is illustrated in Figure 8. If the parameter check is successful, a function call object is created for use in the executable constraint structure.

If the function has any *secret triggers*, then those triggers are added to the parameter list of the function call. The effect of this will be to ensure that each such secret trigger will have a dependents list that includes the constraint now being parsed. One example of secret triggers is the graphical attributes of **`Scale`** and

`Location`, in the case of a function called ConstrainToFillArea(Object, Point). (See the discussion immediately following.)

A method for specifying the non-intrinsic triggers of user-defined constraint language functions and procedures

In an efficiently implemented constraint language execution environment, only the constraints that should be re-evaluated are re-evaluated during each execution cycle. If any of the parameters of an invocation of a function in a particular relation change value, then the relation must be re-evaluated. However, there are some functions that need to be re-evaluated when other values change.

Consider the case of a function called *ConstrainToFill*(Object, Point). This is an iRides function that takes two parameters, a graphical object and an array of two numbers (a point). It returns an array of two numbers that represent the point in the object that is closest to the point specified by the point parameter. (If the point parameter is within the fill area of the graphic, then the same point will be returned. If the point is outside the graphic, then the returned value will be the point on the edge of the graphic that is closest to the point parameter). Naturally, any constraint that invokes ConstrainToFill should be evaluated whenever a parameter value, such as the specified point, changes. However, there are other conditions under which such constraints should be evaluated. If a graphical attribute of the object specified by the first parameter, such as its location or its scale or its rotation, changes, then the relation should be reevaluated.

Users who wish to introduce a new 'native' function or a new 'native' procedure to a constraint language must specify the non-intrinsic (non-parametric or *secret*) triggers of the new function. Changes in the values of *external* attributes, which are not in the function's parameter list, must trigger evaluation of constraints that employ the function. Users override the secretTriggers function of their new descendant of the constraint language function base class in order to specify the external triggering attributes.

The method for adding secret triggers to the parameter list of a function call, referenced in Figure 5, is illustrated in Figure 6.

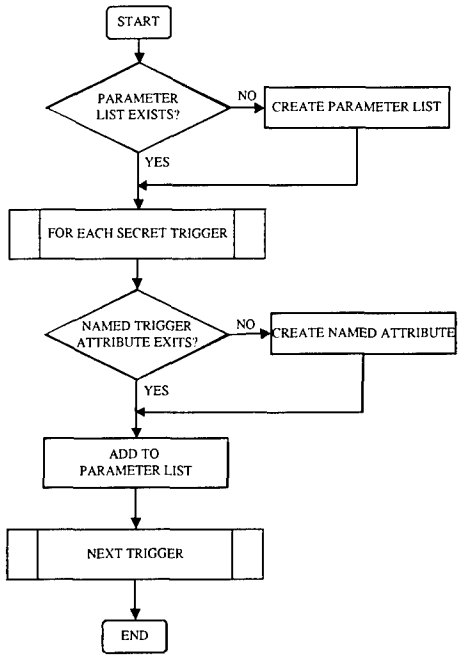
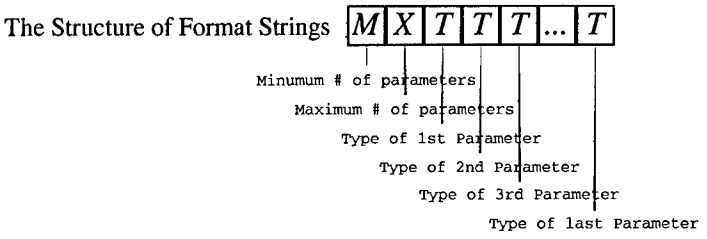


Figure 6. Adding Secret Triggers to the Function Call's Parameter List

If a parameter list does not exist, then one is created, so that the secret triggers can be added to it. The user's secretTriggers() method for this particular function class (which must be derived from a base class with a default secretTriggers) is then called. For each secret trigger in the array of strings that is returned, the method checks for an attribute of that name. If one does not exist, it creates the attribute and assigns it a default value. The set of attribute references for all the secret triggers is then appended to the parameter list for this function call.

Specifying the number and data types of the parameters of a user-defined constraint language's functions and procedures

Users who wish to introduce a new 'native' function or a new 'native' procedure to a constraint language must specify the type and number of the parameters to the new function. This specification takes a very simple form. The getFormat() function is overridden to return a string that consists of a sequence of characters from a restricted set of characters. See Figure 7.



Examples

Format String: 2 2 s n Exactly 2 parameters, 1 string & 1 number

Legal Params: ("engaged", 1)

Format String: 1 1 s , 1 1 3 Either one string or an array of 3 numbers

Legal Params: ("Dodger Blue")
([0.0235, 0.38, 1.0])

Format String: * * * Any number of any parameters

Legal Params: ("Hi Mom!")
([0.0235, 0.38, 1.0])
(0, 1, "pizza", true, 17)
()

Figure 7. The Structure of User-Specified Parameter Format Strings

The first two characters are digits which specify the minimum number of parameters that must be present and the maximum number of parameters that are allowed. The symbol "*" in either of these two positions in the sequence of characters specifies "any number of parameters". The remaining characters in the string returned by getFormat() specify the types of the parameters. Each character in this part of the returned string (in the third to nth position) uniquely specifies the type of the parameter at that position in the list. For example, in iRides, the type specifications are:

- *n* number
- *s* string
- *l* logical
- *o* object
- *N* indefinite length array of numbers
- *S* indefinite length array of strings
- *L* indefinite length array of logicals
- *O* indefinite length array of objects
- *3* array of 3 numbers [any of digits 2..9 OK]
- * any type

If a function can accept several different parameter sets, the parameter specifications must be separated by commas. For example, in the iRides program that uses this technique, the function MakeColor can have a parameter list that

consists of either an array of three numbers (specifying the values of the red, green, and blue components of the color) or a string that names a predefined color. E.g.,

```
makeColor ("Dodger Blue")
makeColor ([0.0235, 0.38, 1.0])
```

These two options can be specified by defining makeColor's getFormat() function as follows:

```
public String  getFormat()      { return "11s,113"; }
```

The designations *n*, *N*, *s*, *S*, etc. correspond to the data types of the iRides constraint language. Here are some examples of particular definitions for getFormat() in different user-defined functions:

```
"44nslo":  4 parms: 1st=number, 2nd=string, 3rd=logical, 4th=object
"****":    Any number of any type (wild card)
"2*ss*":   At least 2 strings and any number other
"44*sn*":  4 parms: 1st and 4th anytype, 2nd string 3rd num
"11s,11n": 1 string or 1 number ("or" is implicit)
"3323N":   3 parms: vector[2].number, vector[3].number,
vector[*].number
"3323O":   3 parms: vector[2].objref, vector[3].objref,
vector[*].objref
```

More formally, the format of the getFormat specifier string (the one parameter to getFormat) is **MXTTT**..., where

```
M is the minimum number of parameters that the function requires
   X must be in one of [* | 0..9]
X is the maximum number of parameters that the function can take
   Y must be in one of [* | 0..9]
T is a type specification
   in iRides, T must be one of [n, s, l, o, N, S, L, O, *, 2..9]
```

How parameters are checked in user-defined constraint functions

A utility object, the parameter checker (a class called ParmChecker in iRides) implements a function that is used to check particular usages of the programmer's own defined constraint language functions. In iRides, this function is called ParmsOK. This function compares the parameters of each usage with the parameter specification provided by getFormat(). See Figure 8.

First the secret trigger references of a user-defined function call are checked. If there are bad references, an error message is logged for the author of the constraint environment. Then a copy of the parameter list is created that does not have the secret triggers. The *getFormat*() function is used to get the user's defined format string, which is used to judge the well-formedness of the parameter list of the function call. This format string may have more than one legal set of parameters specified for the function (comma-separated, as in the second example in Figure 7). For each such parameter specification, the method checks that the number of parameters in the actual parameter list is less than or equal to the minimum number specified by the format string, that the parameter types of the parameter list's actual parameters match the types specified, and that the list does not have more than the maximum number of legal parameters for this specification. If any of these tests is failed, the method moves on to the next specification in the format string and tries again. As soon as a legal match is found, the parameter checker returns success. If all the specifications of the format string are tried without a match, then the Check

Parameters method returns a failure code. In this case, the parameter list supplied in the function call is not legal.

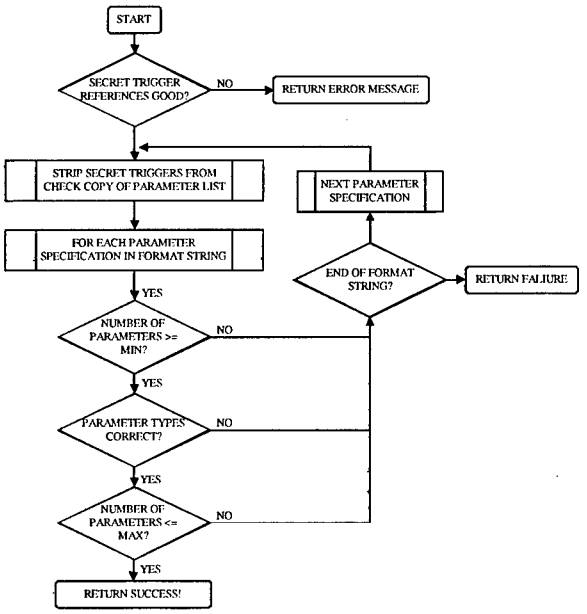


Figure 8. Check Parameters Based on User Specifications

Specifying forced constraint execution for user-defined functions and procedures

In some cases, it may be necessary to define functions that should be updated on every use, whether or not their parameters have changed. An optimized constraint execution system caches previous results with a constraint. If the values referenced in the constraint have not changed, the old cached result can be used, rather than undergoing the computational expense of executing the constraint again. Some functions, however, should be run each time that a constraint is used. A method is needed to specify for a function that constraints with that function should be evaluated on each use. In iRides, this is handled by an overriden base class function, stayDirty().

Most user-defined functions do not need to override the base class definition of stayDirty(), which is defined as :

```
public boolean  stayDirty()      { return false; } //override for
                                                   // random(), etc.
```

Some constraint language systems do not permit functions that stay dirty to be used in constraints, but rather restrict the use of such functions to procedure contexts (*events* in the terminology of the iRides simulation language system). Programming languages that are both constraint languages and procedural languages, such as

the simulation languages in RIDES and iRides, may adopt either approach. In RIDES, functions that stay dirty are permitted in constraints. In iRides, such functions can only be used in events. In RIDES, a constraint can have a form like

```
twitch = random(range) + base
```

This rule will be fired on every simulation cycle; the value twitch will be continuously reset to various values between base and base + range. In iRides, this constraint would be detected at parse time and marked as illegal.

Functions Index

N

O

P

Q

R

S

T

U

V

W

X

Y

Non-Alphabetical